A REPRESENTATIONAL APPROACH TO KNOWLEDGE AND MULTIPLE SKILL LEVELS FOR BROAD CLASSES OF COMPUTER GENERATED FORCES

THESIS
Larry J Hutson
Captain, USAF

AFIT/GCS/ENG/97D-09

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DTIC QUALITY INSPECTED 4

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

19980210049

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

AFIT/GCS/ENG/97D-09

A REPRESENTATIONAL APPROACH TO KNOWLEDGE AND MULTIPLE SKILL LEVELS FOR BROAD CLASSES OF COMPUTER GENERATED FORCES

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Larry J Hutson, B.S.

Captain, USAF

December 1997

Distribution to be determined by ASC/YW

AFIT/GCS/ENG/97D-09

A REPRESENTATIONAL APPROACH TO KNOWLEDGE AND MULTIPLE SKILL LEVELS FOR BROAD CLASSES OF COMPUTER GENERATED FORCES

Larry J Hutson, B.S.

Captain, USAF

Approved:

Sheila B Banks	2DEC97
Maj. Sheila B. Banks Thesis Advisor	Date
Eugene Santos Jr.	2DEC97
Dr. Eugene Santos, Jr. Committee Member	Date
Martin R Stytz	2 Dec 97
Lt. Col. Martin R. Stytz Committee Member	Date

Acknowledgements

My name may appear as the author of this work and I was ultimately responsible for putting "pen to paper", but there's a whole gaggle of people who deserve as much recognition as I. In no particular order, they are:

Dan Zambon, system administrator extraordinaire, *diehard* Green Bay Packer fan, and fervent clean lab policy enforcer. Thanks for the support...and for letting me not only touch but actually wear your cheesehead!

Dave Doak, another system administrator extraordinaire. I appreciate the fact that the words "user error" never escaped your lips...at least, not so I could hear them.

Matt Wilson, a good friend who joined me in countless "stress relieving" outings to Q-Zar and Laser Quest. I promise, now that I'm done, that I'll turn my attention to that computer game we've talked so much about...

Dave Van Veldhuizen and Scott Brown (soon-to-be Drs. Van Veldhuizen and Brown), good friends who never failed to offer advice and wisecracks in appropriate doses. Sorry I won't be able to join you as a Ph.D. candidate...yet.

Kevin Lee, class leader and mentor. Thanks for the many reality checks, morale boosts, and professional development sessions. When I grow up, I want to be the Air Force officer you are.

All the AFIT professors I had the privilege of learning from, among them Lieutenant Colonel Gregg Gunsch, Major Michael Talbert, and Dr. Gary Lamont. Despite the turbulence of AFIT's near closure, not one of you lost focus of why you're here: the students. Thanks.

Eric Loomis, Jim Hooker, and Rob Nix—Ball Aerospace contractors...and perhaps the best group of contractors I've yet worked with during my Air Force career. Gentlemen, I don't care what your statement of work was...you far exceeded it!

Lieutenant Colonel Martin Stytz and Dr. Eugene Santos, Jr., committee members. Bet you never though I'd thank you for pushing me as hard as you did...but here it is.

Major Sheila Banks, my advisor. Thanks for the guidance and mentoring you provided during our many meetings. By the way, you were right; it was a thesis *defense*, not a presentation...

My daughter, Jessica Nichole, who couldn't help but laugh at "a 30-year-old man who's still in school"...and always managed to finish her homework before I finished mine. Between the giggles, kiddo, I hope you see that learning is something you're *never* too old to do.

and, last but *never* least,

My incredible wife, Patricia. Patty, I can never express how deeply grateful I am for your love and support. Your concern for my well-being during the last year-and-a-half touched me and inspired me in countless ways. I'll gladly spend the rest of our life together attempting to repay you in kind. You have all my love, always.

Larry J Hutson

Table of Contents

List of Figures

List of Tables

Abstract

Current computer generated forces (CGFs) in the "synthetic battlespace", a training arena used by the military, exhibit several deficiencies. Human actors within the battlespace rapidly identify these CGFs and defeat them using unrealistic and potentially fatal tactics, reducing the overall effectiveness of this training arena. Simulators attached to the synthetic battlespace host local threat systems, leading to training inconsistencies when different simulators display the same threat at different levels of fidelity. Finally, current CGFs are engineered "from the ground up", often without exploiting commonalities with other existing CGFs, increasing development (and ultimately training) costs.

This thesis addresses these issues by proposing a domain-independent design methodology and a supporting software architecture for the Distributed Mission Training Integrated Threat Environment (DMTITE). This architecture uses approaches from software engineering and database management and identifies an extensible knowledge representation to support CGFs in various domains (land, surface, and air), shifting development efforts from "structure implementation" to "knowledge implementation." CGFs developed using this paradigm also have access to domain-independent features such as skills vectors and a combat psychology model, which act as a time-limited Turing test by making CGF behaviors unpredictable (but not random) and believable.

A REPRESENTATIONAL APPROACH TO KNOWLEDGE AND MULTIPLE SKILL LEVELS FOR BROAD CLASSES OF COMPUTER GENERATED FORCES

I. Introduction

To make better use of limited Department of Defense training funds, the military has turned to Distributed Interactive Simulation (DIS) technology and concepts such as the Joint Synthetic Battlespace (JSB) as training arenas. These concepts and technologies, although successful in certain areas, exhibit several deficiencies resulting from participant and environment inconsistencies.

A major problem is the threat environment (e.g., anti-aircraft artillery or aircraft) currently presented to simulation participants is not consistent. This inconsistency prevents battlespace combatants from interacting in a realistic fashion because they sense the environment at different fidelity levels. For example, simulators of combat force aircraft have native threat generation systems for stand-alone and network testing. This heterogeneous capability often results in compatibility problems between simulators when adding new weapons or theaters of operation. A contributing problem is many battlespace systems representing the same asset are implemented in a non-uniform manner—that is, they are built "from the ground up." Many simulator developers create unique systems and implement modeling decisions supporting their users in specific scenarios. These differing systems are difficult to coordinate on a distributed network because they can not perform at the same fidelity level, creating asset synchronization issues.

Lastly, there is a lack of "intelligent" computer-generated actors within most current military simulations. Humans are able to rapidly identify and defeat current computer-generated entities using unrealistic tactics (i.e., "gaming the system"), reinforcing potentially fatal behaviors. The current situation detracts from the effectiveness of training conducted in virtual training arenas such as the "synthetic battlespace."

Taken together, these issues lead to great expense, uncoordinated threat development, and training inconsistencies between participants. A promising technology to fill this gap is Computer-Generated Forces (CGFs)—computer controlled actors representing combatants and displaying behavior based on current battlespace state information. Large numbers of simulation CGFs could be created on demand with minimal cost, and can be standardized in their presentation of and reaction to threats throughout the battlespace.

1.1 Distributed Mission Training Integrated Threat Environment (DMTITE)

Research discussed in this thesis was conducted in support of DMTITE, a proposed training environment that will support aircrew training by inserting a variety of accurate and realistic threats into large-scale distributed virtual environments. The system will be hosted on one or more computer systems; therefore, each must be able to operate autonomously and also cooperate with other DMTITE systems to portray a coordinated threat environment. Each DMTITE system will require access to a common script for each training scenario and will inject threats (hostile aircraft, radar, electronic countermeasures (ECM), surface-to-air missile (SAM) sites, and anti-aircraft artillery (AAA) batteries) into the synthetic battlespace. Since the DMTITE system will replace the threat generation systems currently local to each simulator, human actors operating in the same domain of the synthetic battlespace will perceive a threat at a common level of fidelity.

This research is sponsored by the United States Air Force's Aeronautical Systems Center (ASC), Air Force Materiel Command (AFMC), Wright-Patterson Air Force Base, Ohio.

1.2 Scope

This thesis addresses two of the goals of the DMTITE project. First, it identifies a knowledge engineering approach for instantiating the broad classes of actors maintained by the DMTITE system. This approach addresses many of the requirements identified during the course of the DMTITE project, but allows developers flexibility in the tools they use to implement DMTITE actors. Second, this thesis identifies the domain-independent software architecture developed to support DMTITE knowledge engineering efforts. This

architecture extends a previously implemented architecture, attempting to shift the CGF development focus from "structure implementation" to "knowledge implementation."

1.3 Overview

Chapter II frames this research by exploring much of the background of the DMTITE project. Terms used throughout this thesis are defined, related efforts are identified, and overviews of relevant concepts are presented. Chapter III identifies a "knowledge-centric" design methodology for implementing CGFs. The assumptions, goals, and components of this design methodology are discussed, and its strengths and weaknesses are identified. Chapter IV maps the design methodology discussed in the previous chapter to a domain-independent software architecture. The components of the architecture are decomposed and discussed in detail. Finally, Chapter V identifies the expected results of this paradigm as well as avenues for future, related research.

II. Background

This chapter introduces the topics of distributed virtual environments (DVEs), including the Distributed Interactive Simulation (DIS) protocol and the High-Level Architecture (HLA); current computer generated force (CGF) background and relevant projects; and the Common Object Database (CODB) system architecture. These topics form the groundwork for the DMTITE architecture and its operational environment. The topic of knowledge acquisition is also discussed since it applies to the research addressed in this thesis.

2.1 Frequently Used Terms

Many of the terms used throughout this thesis are standard within the artificial intelligence and simulation communities; however, the following terms are defined explicitly for those readers unfamiliar with them.

2.1.1 Entities. An *entity* is a component of a distributed virtual environment whose state can change. Combatants (whether human- or computer-controlled) are entities; terrain (whose appearance and features can be changed as a result of plowing, explosions, or traffic) is also an entity.

2.1.2 Actors. An *actor* is an entity that moves with apparent intelligent purpose. Actors can be *virtual* (human-controlled), *constructive* (traditional simulation controlled), *live* (derived from instrumented range data), or *computer-generated* (controlled by a computer program employing artificial intelligence techniques). Obviously, all actors are entities; however, not all entities are actors (i.e., terrain). Computer generated forces are computer generated actors representing combatants in a virtual battlespace; for the purposes of this thesis, the two terms are considered synonymous.

2.1.3 Hosts. A *host* is a computer system within a DVE that allows human and/or computer user(s) to control actors or entities within the distributed virtual environment. A host also allows its users to observe the actions of other entities within

the DVE, whether the other entities are hosted on the same system or another (possibly remote) computer system.

2.2 Distributed Virtual Environments

The DoD realized the usefulness of distributed virtual environments in the early 1980s, when the Army networked its tank simulators through SIMNET (36). Although successful in providing a DVE for armor and other ground vehicles, SIMNET remained limited to that domain. In 1989, the DoD initiated the Distributed Interactive Simulation protocols and in 1995 initiated the High Level Architecture. These two network technologies are the most widely used for DVEs today.

2.2.1 Distributed Interactive Simulation. The DIS suite of standards (IEEE Standard 1278) was designed to link distributed, autonomous hosts into a real-time distributed virtual environment. In DIS, this is accomplished through a network that exchanges data describing events (such as collisions, weapons firings, and detonations) and activities (such as the movement of an actor through the virtual environment). DIS is the epitome of an asynchronous network: there is no central computer, event scheduler, clock, or conflict arbitration system. Stytz provides additional information regarding DVEs and DIS (33), as does Blau (5, 6).

2.2.2 High-Level Architecture. As the heir apparent to DIS, the High-Level Architecture (HLA) takes a more comprehensive approach to communication and basic system requirements. The stated goal of HLA is to establish an architectural framework supporting interoperability between different simulations. A central architectural decision supporting this goal is the separation of *application functions* (managed by a host application software system) from *communications functions* (managed by the Runtime Infrastructure, or RTI). The RTI manages communication paths between executing applications, ensures its applications acquire the data they subscribed to, and publishes data other applications request. The RTI "publish and subscribe" mechanism reduces the amount of data transmitted between applications to only that requested by the applications. The foundational

papers for HLA are contained in the *15th Workshop on Standards for the Interoperability of Distributed Simulations* (8, 11, 14, 24, 32).

2.3 Computer Generated Forces

Computer generated forces (CGFs) are computer-controlled actors in a distributed virtual battlespace. In general, CGFs attempt to model either human cognition or human behavior in combat; approaches to achieving realistic CGFs are described by Calder, et al. (7), Edwards (13), Laird (19, 20), and Tambe (35). The runtime challenges for a CGF arise from the need to compute human behaviors and reactions to a complex dynamic environment. Other research (such as TacAir-Soar) addresses this challenge by attempting to *emulate* the human cognitive process (19). On the other hand, this research assumes that *simulating* the observable aspects of human decision making is sufficient. This assumption somewhat eases the computational requirements of a CGF.

Unfortunately, many rapidly developed CGFs fail to display realistic and accurate outputs when compared to human counterparts. This modeling deficiency allows human-controlled actors to easily identify their computer-controlled counterparts, thus yielding an unrealistic advantage to and reinforcing potentially fatal behaviors in the participants being trained. Both the physical and mental representations of a CGF must be realistically modeled to ensure realistic outputs. Karr, et al., present an overview of the requirements for and current deficiencies in CGF representations (18); Santos, et al. (27), and Rosenbloom, et al. (25), present approaches for developing "human-like" CGF mental representations.

2.4 Common Object Database Architecture

The Common Object Database (CODB) is a data-handling architecture that uses object classes, containerization, and a central runtime repository to manage and route data between applications in a distributed virtual environment (34). There can be multiple CODBs on a single host, most of which will contain only the information required by the applications directly connected to them. One CODB, however, must maintain the entire current state of the DVE via a "world state manager" (WSM). The WSM sends and receives networked information about entities in a distributed simulation; calculates entity positions

via dead reckoning between updates; and converts network messages to their corresponding CODB container data structures. It is the responsibility of each subordinate CODB to ensure the appropriate information propagates both to and from the WSM CODB.

Stytz, et al. (34) describes the CODB architecture. The CODB approach and implementation used in support of this research are discussed in Appendix A.

2.5 Knowledge Acquisition

Knowledge acquisition consists of two tasks, both essential to the development of knowledge-based systems such as CGFs. During *knowledge elicitation*, a knowledge engineer gathers knowledge from sources such as domain experts, books, reports, and visual inspection (15). Once a sufficient amount of knowledge is obtained, a *knowledge representation* is used to convert it to a computer-readable format. This process is often iterative, and is discussed in great detail by Gonzalez and Dankel (15). Examples of knowledge acquisition techniques for CGFs include "semantic areas of concern" used by Zurita (38) during the Intelligent Wingman project, and "storyboarding", utilized by Banks and Lizza (21) to implement the Pilot's Associate.

III. DMTITE Design Methodology

DMTITE is designed to allow human pilots to train against a variety of threat systems, some of which operate in the same domain as the pilots, some of which do not. For example, opposing aircraft operate in the same domain as human pilots, while land-based radar and electronic countermeasure (ECM) systems do not. Further complicating matters are threats that operate in multiple domains, such as surface-to-air missile (SAM) and anti-aircraft artillery (AAA) sites—land-based entities which generate airborne entities (e.g., missiles and bullets). CGFs simulating these threats require different knowledge, tactics, and physical models to accurately portray a realistic threat environment. Current CGF development efforts build entities "from the ground up", factoring the domain of interest into early design decisions, resulting in a software architecture tightly coupled to that domain. Commonalities between CGFs are often overlooked since similar CGFs are often viewed as having distinct and different roles and responsibilities within the virtual battlespace. The use of proprietary software development tools and practices further complicates matters since they are not transferable to other CGF development efforts. The results of these practices are higher development—and ultimately training—costs, as well as a lack of coordination between CGFs.

Identifying a design methodology to rapidly design, implement, and test DMTITE CGFs addresses these concerns in several ways. First, a design methodology defines a standard approach for developing CGFs, addressing the software engineering concept of *reuse* by encouraging developers to evaluate new requirements in terms of existing CGF implementations. As a result, work duplicated among multiple CGF development efforts is minimized. Second, a design methodology serves as a roadmap, helping both knowledge and software engineers identify the type and scope of work to be accomplished. This allows CGFs to be developed in a more effective and efficient manner. Finally, identifying a design methodology allows a software architecture to be implemented in support of it. This architecture becomes a tool of the methodology, shifting development efforts from *structure implementation* (concerns such as processing flow and interprocessing communication) to *knowledge implementation* (the specific knowledge and state information required by a CGF). Actual software development is then limited to designing and implementing physical

models not yet incorporated as part of the software architecture, resulting in a more efficient manner of developing CGFs.

3.1 Goals of the Design Methodology

The first goal of the DMTITE design methodology was to avoid specifying knowledge elicitation and documentation techniques to be used. The design methodology needed to be flexible enough to allow knowledge engineers to use whatever techniques are available and appropriate. Another goal of the DMTITE design methodology was to minimize the amount of duplicate knowledge used to implement a CGF. This goal is similar to the concept of *centralized control* in database systems (12); it controls redundancy, helps avoid inconsistencies in the knowledge, and maintains knowledge integrity. A third goal was to identify a taxonomy capable of supporting a domain-independent approach to developing CGFs while acknowledging the fact that each domain requires knowledge not shared by other domains. A final goal of the design methodology was to incorporate the identification of "knowledge modifiers" into the CGF development cycle. With such wide ranging goals, then, the ultimate goal of this design methodology is to identify components essential to the development of a CGF (regardless of domain), then to bring those components together in a cohesive, usable manner.

3.2 Knowledge Representations

While the design methodology assumes a suitable method of eliciting and documenting knowledge to construct a CGF exists, at some point that knowledge must be mapped to a single representation supporting multiple inferencing strategies. A single representation allows knowledge to be accessible to *any* inferencing strategy with only minimal modification, reducing the work required to inference over knowledge originally implemented for some other inferencing strategy. Furthermore, this representation must support the goal of minimizing the amount of duplicate knowledge in the system. This allows knowledge to be added, deleted, or modified in an efficient and effective manner. Finally, while not an explicit goal of the design methodology, the knowledge representation shouldn't prevent the knowledge from being dynamically added, removed, or modified by the CGF.

3.2.1 Knowledge Expressions. At the lowest level, the knowledge representation must be able to express knowledge in the form of rules, facts, and so forth in a manner usable (or at least decipherable) by an inferencing strategy. For DMTITE CGFs, three possible inferencing strategies were identified: case-based reasoning, rule-based reasoning, and fuzzy logic. Since each of these inferencing strategies have been demonstrated to be computable, they can be derived from *primitive recursive functions* (22); that is, they are derived from a common set of predicates. These predicates can be expressed as *if-then* (or *if-then-else*) expressions; as a result, these terms are used to express the knowledge used by any of the inferencing strategies supported by DMTITE.

However, the traditional variables contained in most *if-then* terms do not fully support fuzzy logic. A knowledge representation supporting fuzzy logic must support not only "crisp" (traditional) variables, but also "fuzzified" variables (representing a range of values). Since this is the only difference between a traditional *if-then* term and a "fuzzy" *if-then* term, an extensible knowledge representation is readily identifiable and implemented. (The knowledge representation used in DMTITE is discussed in Appendix C).

3.2.2 Policies: "Atomic" Knowledge Bases. At a higher level of abstraction, the knowledge representation must support knowledge bases comprised of smaller, more "atomic" knowledge bases. Borrowing a term coined by Earl Cox in support of his Fuzzy Modeling System (10), these "atomic" knowledge bases are *policies*—logical and self-contained units of knowledge. This knowledge representation has two significant advantages. First, it supports the goal of minimizing the amount of duplicate knowledge in DMTITE. For example, a given aircraft CGF might define an "offensive posture" knowledge base as consisting of an "offensive maneuvers" policy and a "known enemy maneuvers" policy, while defining a "defensive posture" knowledge base in terms of a "defensive maneuvers" policy and the *same* "known enemy maneuvers" policy (Figure 3.1). Additional enemy maneuvers can then be added to a single policy while being reflected in both knowledge bases. The other advantage of this knowledge representation is *knowledge abstraction.* Just as data abstraction allows use of an object without knowledge of its underlying data structure, knowledge abstraction allows an entity to access a knowledge base seemingly

common to other CGFs, but consisting of different policies. For example, CGFs may search for threats using optical sensors ("eyes"), radar, or both. A general search knowledge base can be defined consisting of policies specific to the sensors available to a given CGF (Figure 3.2). Furthermore, the general search knowledge base can be incrementally implemented and tested by including each of its constituent policies one at a time.

3.2.3 Knowledge Repository. At the final level of abstraction, the knowledge representation should allow knowledge to be centralized. This centralized nature allows knowledge to be globally available to all inferencing engines requiring it, but doesn't require each engine to maintain a local copy of the knowledge it uses. Centralization of knowledge also allows engines to add, delete, and modify knowledge bases in a manner that makes such changes available to all engines using those knowledge bases. If an inferencing engine requires a local copy of the knowledge (for instance, the knowledge representation is not directly supported by the inferencing strategy), this approach permits such copies to be made. The flexibility of a knowledge repository makes the design methodology more attractive to both knowledge and software engineers.

3.3 Knowledge "Modifiers"

CGFs should display multiple skill levels in a manner similar to those displayed by human actors and the design methodology must identify a means of capturing this information. Since the presence of skills is reflected in the behavior of a combatant, and the behavior of CGFs is based on the knowledge available to that CGF, skills are essentially "knowledge modifiers." For example, an anti-aircraft artillery CGF with little weapons skill may display this by firing at targets out of range, while an identical CGF with more skill would wait until the target is within weapons range. While eliciting and documenting knowledge, the knowledge engineer should identify the presence of skills in the domain experts. Differences between the actions described by a domain expert and the actions dictated by doctrine are one example of the presence of skills. Unfortunately, no one method can identify the presence and effect of skills; as a result, the knowledge engineer must carefully analyze acquired knowledge and determine if skills affect it.

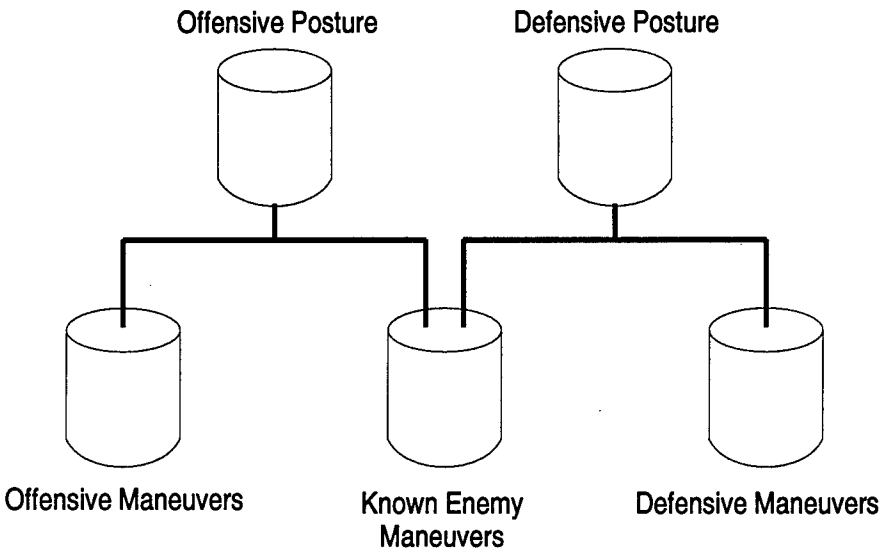

Figure 3.1 Shared Policy Example

How skills affect knowledge is no easier to identify. In most cases, the skills will indirectly affect knowledge by modifying the information that triggers it. Returning to the previous example, the unskilled AAA CGF *extends* the firing range of its weapon beyond its actual limit. One way of reflecting this is to determine the farthest range an unskilled operator would fire at and derive a mathematical function that reduces the range as skill increases. Another approach would be to determine the actual firing range of a weapon, then derive a mathematical equation that extends that range as skill decreases. Again, how the skill ultimately affects the knowledge is left to the judgment of the knowledge engineer; however, the ultimate goal here is to reflect actual observations into the synthetic battlespace.

3.4 A Supporting Taxonomy

An approach that assumes a single cognitive representation for multiple CGFs implies a taxonomy exists for the CGFs in question. In the case of distributed virtual environments, the DoD developed such a taxonomy, albeit embedded in the DIS standard (IEEE 1278.1-1995). In DIS, the "Entity State" Protocol Data Unit (PDU) defines (among other things) the category to which a CGF belongs (17). This information is the ideal basis for a

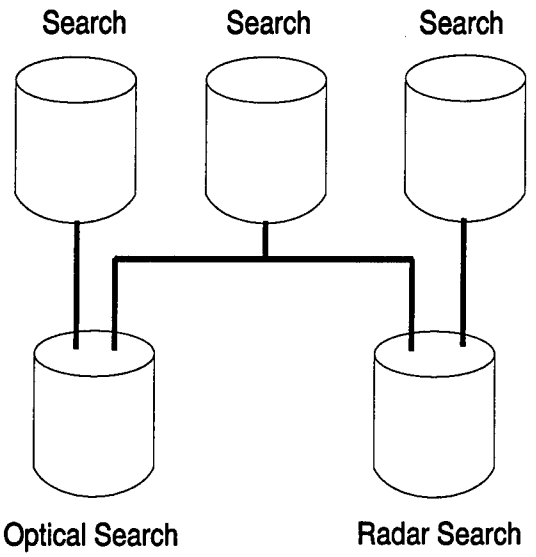

Figure 3.2 Knowledge Abstraction Example

taxonomy, if one accepts these categories as defining the various mission platforms within the land, surface, air, and space domains.

Although DMTITE is capable of supporting space-based CGFs, this domain is notably missing from the taxonomy. The DIS standard includes a very limited taxonomy of space-based entities, dividing them into two categories ("manned" and "unmanned"), and enumerating only the U.S. Space Shuttle fleet under the "manned" category. The DIS taxonomy (and the corresponding DMTITE taxonomy) reflect the current political agreements banning space-based weapon systems. Obviously, if the political environment changes, a taxonomy for the space domain will need to be implemented.

3.5 A "Knowledge-Centric" Design Methodology

The design methodology (discussed in detail in Appendix B) resulting from the incorporation of the components identified in this chapter consists of two parts. The first half focuses on what knowledge is necessary to instantiate a CGF category within a given domain (e.g., land, air, surface) in the DMTITE taxonomy. During this phase of development, a knowledge engineer elicits and documents knowledge from the appropriate sources, groups related knowledge into policies, and determines the inferencing strategies to be em-

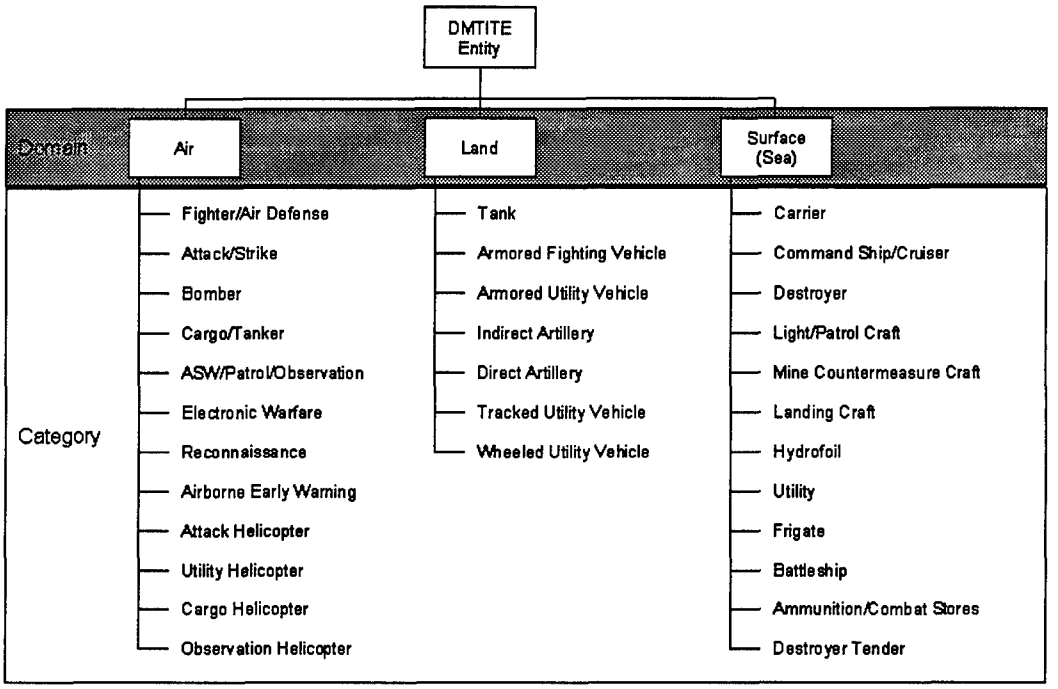

Figure 3.3 Initial DMTITE Taxonomy

ployed by the CGF. While acquiring knowledge, the knowledge engineer also identifies knowledge "modifiers" (expressed as skills and other operator capabilities) and determines how these affect the knowledge in question. When this phase of the methodology is complete, the knowledge engineer has developed the *cognitive representation* for a category of CGFs in the DMTITE methodology. The first half of the design methodology is the most time-consuming aspect of developing knowledge-based systems such as CGFs (15), but should be conducted fully only once for each CGF category. As additional knowledge is identified for inclusion within a CGF category, the knowledge engineer will need to revisit this phase of development, although these efforts should be smaller in scope than the initial development effort.

On the other hand, the second phase of the methodology will most likely occur every time a new CGF type (e.g., F-15, F-16, F-22) within a category is instantiated. In this phase, the knowledge engineer works closely with a software engineer to ensure the proper information is used to invoke the CGF's knowledge. Physical models are identified and, as

necessary, designed, developed, and tested. The information provided by these models is mapped to the information required by the cognitive representation. In short, when this phase is complete, the knowledge and software engineers have identified and implemented the information upon which the CGF will base its behaviors. Since this includes simulation-specific information such as weapon loadouts, fuel levels, and so forth, this phase of the methodology will most likely be performed each time a CGF is to be instantiated for use in a distributed simulation.

This design methodology is "knowledge-centric" since it, in its first phase, is concerned with *what knowledge is required* by a CGF (or category of CGFs) and how that knowledge is modified through skills and other capabilities while, in its second phase, it focuses on *how that knowledge is invoked.* The knowledge and software engineer no longer are concerned with essential but domain-independent issues such as how the CGF communicates its state to the DVE. In fact, the design methodology itself is not concerned with such issues; instead, it assumes these concerns are integrated into the underlying software architecture.

IV. A Domain-Independent Software Architecture

In addition to supporting the design methodology outlined in Chapter III, the DMTITE domain-independent architecture is required to fulfill several other goals. First, the software architecture must be able to support live, virtual, and constructive simulations, since each of these are used in military training. Second, the software architecture must be flexible, so that current CGF development efforts can be used to address future CGF requirements. Third, the architecture must support domain-independent concepts such as combat psychology, communication, and cooperation. The design methodology this architecture supports assumes these issues are addressed by the architecture and, as a result, the architecture must allow such concepts to be incorporated with no effect on the design methodology. Finally, the software architecture must address the concerns of CGF behavior *consistency*, *unpredictability* and *certifiability*. Consistency establishes behaviors reflecting a given state of the virtual environment; in other words, behaviors that are not random or completely ignore the environment state. Unpredictability eliminates exploitable patterns in CGF behaviors. Certifiability measures CGF behaviors against human behaviors in similar situations (4).

This chapter begins by describing a general architecture, the first step in meeting these goals. The general architecture is then extended to encompass domain-independence and further address unpredictability and certifiability. Next, the concepts of the design methodology and the components of the extended architecture are instantiated. Finally, these components are brought together to establish the complete and integrated DMTITE software design.

4.1 An Existing Architecture: The General CGF Architecture

Santos, et al. (27), have proposed a general architecture of CGF components. Figure 4.1 shows this architecture, which consists of a physical dynamics component, an active decisions component, and a CGF router. The key of this architecture lies in the separation of physical and cognitive processes, which allows several advantages. First, it minimizes dependencies between the two processes; changes to one do not necessarily impact the

other. Second, it shifts the focus from *what* knowledge is available to *how* the knowledge decomposes. Finally, it allows instantiated CGFs to display a wide variety of abilities and skills.

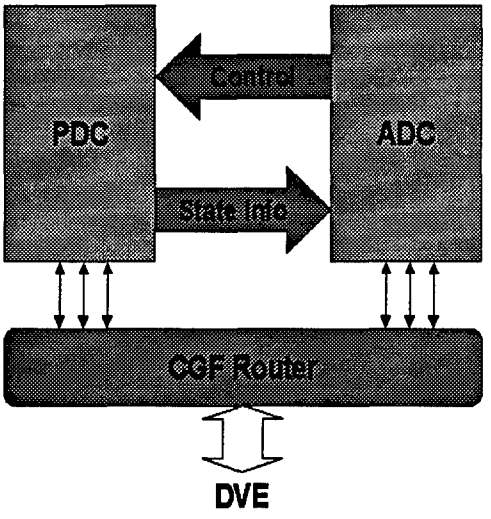

Figure 4.1 General CGF Architecture (27)

4.1.1 Physical Dynamics Component (PDC). The PDC consists of components modeling a CGF's physical aspects: kinematics models, sensor models, weapons models, and so forth. As proposed by Santos, et al. (27), the PDC also initializes parameters that give a CGF a specific identity, such as performance specifications and operator capabilities.

4.1.2 Active Decisions Component (ADC). The ADC consists of four components that use information from the PDC to make decisions. The Strategic Decision Engine (SDE) is concerned with high-level functions such as identifying, implementing, and revising mission goals. The Tactical Decision Engine (TDE) manages the moment-to-moment operations of the entity, such as determining which maneuvers to execute or weapons to employ in a given situation. The Critical Decision Engine (CDE) represents the survival instinct of the entity, determining if an emergency situation exists and (if so) what actions to take. Finally, a Basic Control Module (BCM) converts signals from the decision engines to appropriate physical model inputs.

4.1.3 CGF Router. The CGF router is the interface between the distributed virtual environment (DVE) and the entity. As proposed by Santos, et al. (27), both the PDC and ADC directly access the CGF router.

4.2 Extending the General Architecture

The DMTITE architecture (shown in Figure 4.2), originally proposed by Van Veldhuizen and Hutson (37), extends the general architecture to support both a domain-independent approach to implementing CGFs and address the concern of certifiability (which is only partially addressed by the general architecture). The high-level mapping between the general architecture and the DMTITE architecture is shown in Table 4.1 and discussed in the following sections.

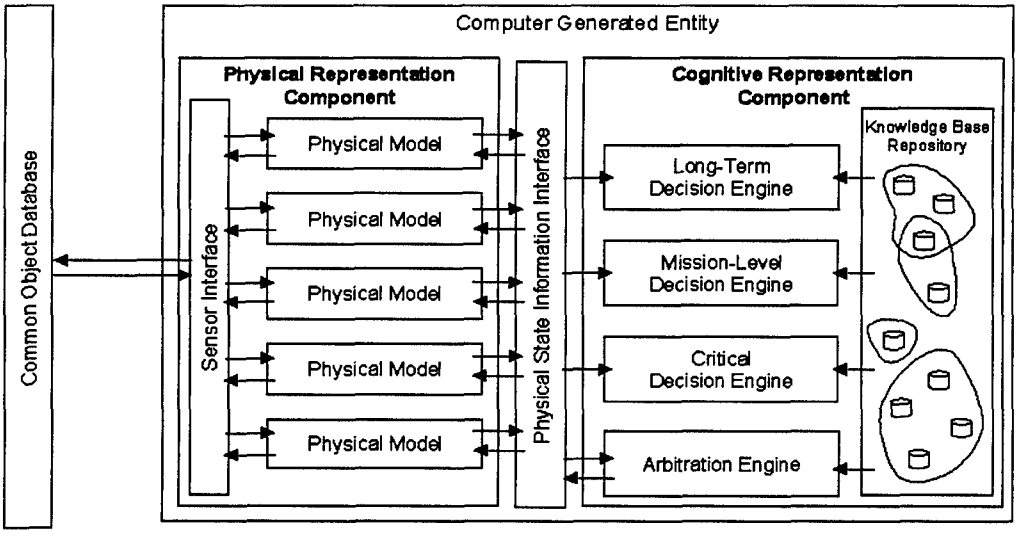

Figure 4.2 DMTITE Software Architecture

4.2.1 Physical Representation Component (PRC). The PRC maps closely to the PDC. Like the PDC, the PRC consists of physical models (such as those identified previously). These physical characteristics are viewed as objects and operators; the resulting modularity allows for easy addition, deletion, and modification of these components without requiring corresponding changes to the cognitive representation.

Table 4.1 High-Level Relationship between General and DMTITE components

General Architecture	*DMTITE Architecture*
Physical Dynamics Component	Physical Representation Component
Active Decisions Component	Cognitive Representation Component
Critical Decision Engine	Critical Decision Engine
Tactical Decision Engine	Mission-Level Decision Engine
Strategic Decision Engine	Long-Term Decision Engine
Basic Control Module	Arbitration Engine
CGF Router	Sensor Interface
(implicitly defined)	Physical State Information Interface

However, the PRC doesn't *exactly* map to the PDC. The PRC is comprised strictly of the physical attributes and properties of the CGF; those characteristics that are subject to physical laws. Concepts such as "skills" and "operator capabilities" don't observe such laws and have been moved to the cognitive representation. Another significant change is that the PRC represents the CGF's sole communications channel to the DVE. This change was made to more closely reflect reality, where humans are forced by design to interact with their environment *directly* through physical processes. In the case of the CGF, interaction is handled by the sensor interface. This component queries the DVE for information in domains of interest (such as land, surface, or air) and relays this information to the physical models. The sensor interface is also responsible for keeping the DVE informed of the entity's existence as well as any events triggered by the CGF by submitting the proper containers to the local CODB.

4.2.2 Cognitive Representation Component (CRC). The CRC encompasses CGF characteristics that are less physical and more cognitive in nature. It is responsible for simulating the *outcomes* of the human cognitive process (decisions), but does not require a model of the cognitive process to be implemented. While, much like the general architecture's CDC, the CRC is concerned with decision-making and also contains the CGF's goals, entity profile, and knowledge bases.

The CRC, much like the CDC, consists of three decision engines, although the names were changed to minimize the semantic impact of words such as "tactical" and "strategic"

(since these terms have distinct meaning within the warfighting communities). The Long-Term Decision Engine (LTDE) reasons over strategic plans and goals of concern to the CGF. For example, it allows the CGF to identify "targets of opportunity" that advance the ultimate goals of the simulated mission, but were not part of the CGF's original tasking. The Mission-Level Decision Engine (MLDE) reasons over moment-to-moment and short-term actions of the CGF. Such actions include, for example, initiating a bomb drop or attacking enemy aircraft. The Critical Decision Engine (CDE) is the only engine that remains unchanged in both name and function from the general CGF architecture. Like the general architecture, the DMTITE architecture has three decision engines scoped to different levels and tasks. This scoping allows fine-tuning of a single set of behaviors (e.g., long-term planning) with minimal impact on other behaviors. Unfortunately, each engine will most likely render a different decision for the same situation.

To overcome this situation, the general architecture's BCM is replaced in the DMTITE architecture with an Arbitration Engine, a specialized decision engine responsible for selecting the decision ultimately enacted. The Arbitration Engines polls the other decision engines and considers not only those decisions, but also each decision's merits relative to the current state of the entity profile. (The entity profile is discussed in detail in Section 4.4.) As a result, the Arbitration Engine can simulate abstract concepts such as "fear" (by selecting the CDE's decision), "bravery" (by ignoring the CDE's decision and choosing the MLDE's), and "indecisiveness" (by choosing to ignore all decisions). In short, the Arbitration Engine ultimately decides the course of action a CGF takes while attempting to simulate human behaviors in those actions.

The final component of the CRC is the Knowledge Base Repository. This repository maintains the sum of the knowledge available to the CGF, minimizing the duplication that would occur if each decision engine maintained local copies of this information. Decision engines access this knowledge at the knowledge base level, having no direct access to the individual policies that comprise each knowledge base. While not part of this research, the Knowledge Base Repository also allows knowledge to be dynamically added, deleted, and modified in one location while affecting multiple decision engines.

4.2.3 Physical State Information Interface (PSII). The PSII is the CRC's sole source of state information. It consists of a variety of *state messages*, data structures that group related physical state information. For example, the state message representing a hostile entity specifies information such as its domain, heading, speed, orientation, and type. Physical models place state messages into the PSII, and the CRC can retrieve related state messages generated by different physical models via a single call to the PSII.

The PSII (as well as the Sensor Interface in the PRC) supports the concept of *data filtering*, shown in Figure 4.3. "Pure" state information is manipulated by the physical models in the PRC to produce "sensor-corrupted" information, which is stored in the PSII. The CRC retrieves this "sensor-corrupted" information from the PSII, corrupting it further by adding the effects of the entity profile. This "sensor- and skill-corrupted information" is the actual information used by the CRC to make decisions.

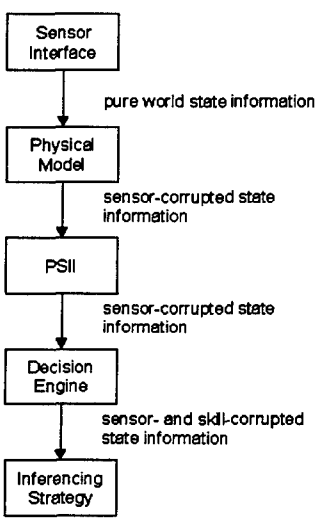

Figure 4.3 Data Filtering in DMTITE

The PSII also serves as the repository for physical model control information. When the Arbitration Engine decides to interact with the DVE, it places the corresponding control messages in the PSII. On the next update cycle, each controllable physical model queries the PSII for any control messages affecting it. Any applicable control messages are acted upon by the physical model. Ultimately, the Sensor Interface gathers the effects

of these interactions and submits the information to the local CODB, which ultimately transmits this information to other actors in the synthetic battlespace. In short, the CRC doesn't *directly* interact with the DVE; instead, it interacts with its physical representation which, in turn, propagates these interactions to the DVE.

4.2.4 Common Object Database (CODB). To ensure compatibility with the current Distributed Interactive Simulation (DIS) protocol as well as the proposed High-Level Architecture (HLA) protocol, the communications aspects of the CGF were "removed" from the CGF and placed in the domain of a Common Object Database. The CODB serves as a repository for information being distributed to and collected from the DVE. CGFs interact with the CODB through the use of data containers: the CGFs are responsible for populating their portion of the container with the appropriate information and routing it to the CODB. The CODB then (conceptually) "repackages" this information to meet the requirements of the communication protocol being used. As long as the CGF is placing the proper information in its containers, the CODB will be able to meet the communications requirements.

Although the CODB concept has been in place at AFIT for several years, it has yet to be fully realized. Several simplifying assumptions have been made regarding its implementation, rendering it not much more than a memory arena utilized by different applications. In support of DMTITE, the CODB concept was revisited and reimplemented (Appendix A). As a result, DMTITE CGFs have access to the full capabilities of the CODB concept, as proposed by Stytz, et al. (34).

4.3 Knowledge Representations

The DMTITE software architecture is ultimately responsible for providing the data structures and implementations of the knowledge representations identified by the design methodology. This responsibility is best addressed by applying software engineering principles to the problem, as discussed in the following subsections.

4.3.1 Knowledge Expressions. As discussed in section 3.2.1, a general knowledge representation based on *if-then-else* predicates is required to support the various inferencing strategies to be used. This representation must be extensible to support features of one inferencing strategy not available (or known) to another, while residing in a common repository. Fortunately, the object-oriented concept of *inheritance* supports these requirements.

The DMTITE software architecture defines a generic *Expression* class from which all knowledge expressions are derived. The *Expression* class is purely virtual; it establishes the access methods to be supported by all derived types, but can't be used to directly instantiate a knowledge expression. The current format for the knowledge expressions used for case-based, rule-based, and fuzzy logic inferencing in DMTITE is discussed in Appendix C.

4.3.2 Policies. The software architecture directly maps the concept of policies (discussed in section 3.2.2) to a data structure. As with knowledge expressions, policies may encapsulate information in one inferencing strategy that is neither known or required by another. In much the same manner as was done for expressions, a purely virtual *Policy* class is defined from which policies specific to each of the inferencing strategies are derived. Unlike the *Expression* class, which doesn't define the contents of derived expressions, the *Policy* class forces all derived policies to initially view the expressions in a generic sense, casting them to the appropriate derived type internally. More information on the case-based, rule-based, and fuzzy logic policy classes can be found in Appendix C.

4.3.3 Knowledge Bases. At this level of abstraction, the software architecture departs from the design methodology. From the methodology viewpoint, a knowledge base is comprised of all *knowledge* in its constituent policies. From the architectural viewpoint, a knowledge base *relies* on the policies to define its constituent knowledge. In other words, the knowledge base class contains a list of policies to be referenced and the methods to access those policies. However, inferencing engines aren't aware of this distinction; they access the knowledge bases as if the knowledge resided within them.

4.3.4 Knowledge Base Repository. The software architecture views the knowledge base repository as a collection of policies (where the actual knowledge expressions reside) and knowledge bases. Policies are not directly accessible to users of the repository; instead, the repository forces users to view knowledge at the knowledge base level.

4.3.5 Type-Independent Variables. One aspect of knowledge expressions not addressed by the design methodology involves variable storage. Variables of different types require different amounts of memory, so a means of abstracting this detail away is necessary. Again, inheritance comes into play.

The software architecture implements a type-independent variable class. This class requires the user to specify the type of data being stored initially, as well as forcing the user to explicitly retrieve the value, but does allow knowledge expressions to store this information. Methods are provided to evaluate relationships (e.g., "less than", "greater than") between variables, and mathematical expressions can be embedded and evaluated within a variable. The impact of this design decision is that a small amount of additional processing time is required to evaluate an expression and additional information must be embedded in each expression (see Appendix C); however, the flexibility provided by this approach far outweighs the costs.

4.4 Simulating Human Behaviors: The Entity Profile

As stated in section 4.2.2, the Arbitration Engine maintains an "entity profile" in support of multiple skill levels. The first component of the entity profile is the skills vector, part of the general architecture proposed by Santos, et al. (27) This vector is supplemented by entity *traits*—variables defined for all CGFs (regardless of their domain) and used to support a combat psychology model. When combined, these concepts address behavior consistency, unpredictability, and certifiability.

4.4.1 Skills Vector. A CGF skills vector is a hierarchical collection of parameters representing the current skill level of the CGF in question. Highly specialized skills are

mapped to more basic skills, both of which may change over time. This interaction between skills guarantees consistency in the behavior and performance of the CGF (27).

The skills vector is a domain-independent concept with domain-specific instantiations. While a skill may be shared among different CGF categories, no skill will be common to *all* categories. As a result, the DMTITE software architecture only defines the skills vector as an object and makes no attempt to define its contents. Instead, the skills to be used by a CGF are defined at runtime via an initialization file (see Appendix E).

4.4.2 Entity Traits. Human behaviors in combat aren't strictly a measure of skill; they reflect other, less tangible concepts. Unlike skills, these concepts (or *traits*) apply to all combatants, albeit in varying degrees due to genetics and life experience (29). When used in conjunction with a combat psychology model, these variables address the unpredictability and certifiability concerns of CGF behavior by modifying the CGF's behavior not only in response to that entity's skills, but also its evaluation of the current environment. For example, consider two CGFs that have identical skill vectors (i.e., they are equally skilled and seemingly identical). If the CGFs traits are not considered, these CGFs will respond identically to the same input—revealing a predictable pattern that can be exploited by human actors. Introducing traits quantifies the less tangible attributes of these CGFs, and allows (for example) one CGF to hesistate in a given situation while the other CGF does not. Although two CGFs with identical skills vectors *and* traits will display the same behavior in a given situation, the addition of these extra variables makes exploitable patterns more difficult to discern.

4.4.3 Combat Psychology Model. If human behaviors are to be simulated, then human behaviors under combat situations must be incorporated into the entity profile. Current CGFs tend to "fight to the last man" with little or no effect on their performance, forcing human controllers to order heavily damaged CGFs to retreat (18). These behaviors undermine the *unpredictability* of CGFs, since they establish patterns of behavior that can be exploited by human actors. Furthermore, these behaviors aren't *certifiable* since even highly trained combatants suffer performance degradation under harsh battlefield conditions.

Incorporating a combat psychology model in the architecture addresses these concerns. CGFs operating under such a model can suffer progressively increasing performance degradations, from slight hesitations to panic-stricken retreat. The combat psychology model incorporates multiple levels of behaviors in a manner identical to how the skills vector supports multiple skill levels. Therefore, the combat psychology model also supports *consistent* CGF behaviors, since a given CGF "psychological profile" displays behaviors consistent to those displayed by a human combatant with a similar psychological makeup. The combat psychology model used in the DMTITE is discussed in detail in Appendix D.

4.5 Bringing It All Together: The DMTITE Entity Design

The concepts and components discussed in the previous sections are brought together in the initial DMTITE software design (Figure 4.4).

This *object model* represents the structural aspects of a DMTITE entity, establishing the objects and the relationships between objects that comprise the DMTITE software architecture. Objects that share common structure and behavior (i.e., decision engines) are represented hierarchically. Since the structure of a DMTITE entity is defined by this architecture, software engineers can implement decision engines to support specific inferencing strategies *without concern as to how the engine will be invoked*—the processing flow already addresses this concern. Additionally, existing physical models can be incorporated into DMTITE by simply ensuring each uses the interface defined by the object model. Assuming the existing physical model is itself modular in design, little (if any) modification will be required to the model itself.

The object model also incorporates many domain-independent concerns while not explicitly declaring them. For instance, the entity profile is itself nothing more than a collection of variables used by the Arbitration Engine. This engine is derived from a generic *Decision Engine* object, which already includes a set of variables. As a result, the entity profile is subjected to *abstraction*; as an architectural component it is unimportant and therefore suppressed. This allows knowledge engineers to identify and software engineers to implement the entity profile *as nothing more than a list of variables.* The architecture handles these variables no differently than any other.

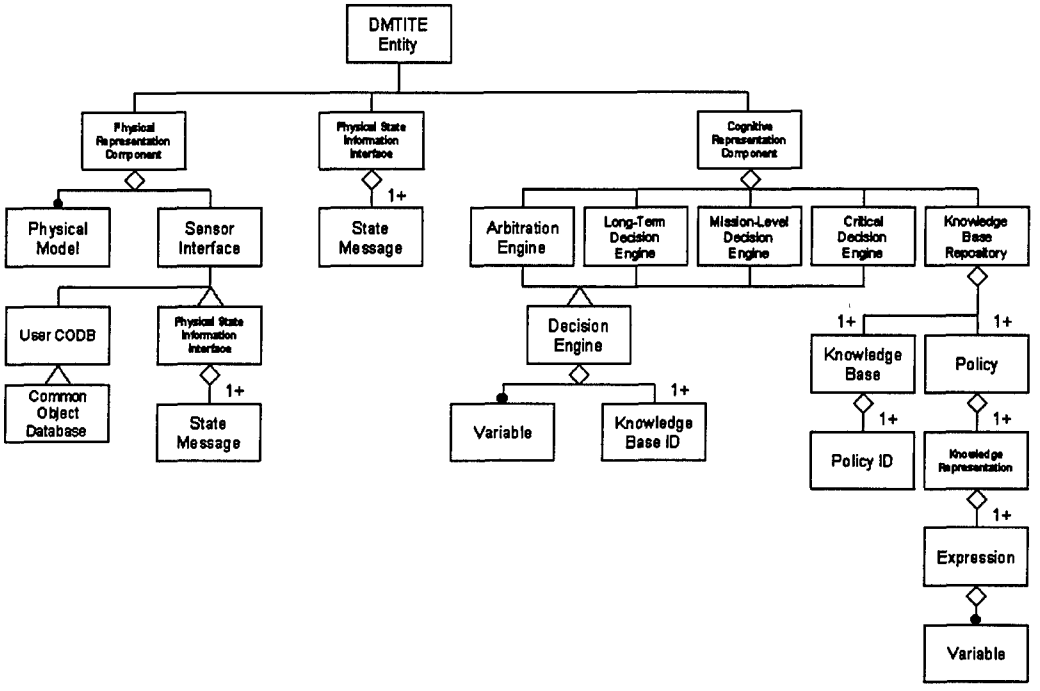

Figure 4.4 DMTITE Entity Design

Finally, the object model defines how state information is passed throughout a DMTITE entity. The CRC can not "go outside" this model to obtain information directly from the distributed virtual environment. This is because the *Cognitive Representation Component* object is physically isolated from the *Sensor Interface* object. On the other hand, the architecture allows physical models to retrieve state information from or pass state information to *either* of the state information repositories (the Sensor Interface and the Physical State Information Interface). Furthermore, since the only difference between the two repositories is the "flavor" of information each contains, the object model provides one set of methods for both. This allows software engineers to modify how either decision engines or physical models send and receive information *without requiring them to learn two different sets of methods.*

In short, the DMTITE object model allows both knowledge and software engineers to implement CGFs without concern to the underlying architecture. Assuming the required physical models have been previously incorporated into DMTITE, new CGFs can be im-

plemented solely by identifying the knowledge required and how that knowledge is invoked. If a physical model is required that hasn't been previously implemented in DMTITE, only minimal interaction with the software architecture is required.

V. Contributions and Recommendations

5.1 Contributions

The design methodology and software architecture developed during the course of this research contribute to future CGF development efforts in many ways. The domain-independent nature of the design methodology allows it to be used not only during the development of DMTITE CGFs, but also for other CGF development efforts. Knowledge engineers can incorporate new knowledge acquisition tools and techniques in the methodology as they become available. Software engineers can develop models to reflect evolving weapon systems, or new decision engines to address various inferencing strategies, without detracting from the overall methodology. The extended general architecture has been implemented using accepted software engineering techniques. This research has blurred the line between the architecture (a series of black boxes that define *what* processing must be done) and the software design (which defines *how* the black boxes accomplish their tasks). Knowledge engineers can add knowledge to this architecture and gauge the effects of their additions without knowledge of the underlying software architecture. Software engineers can implement customized black boxes with no concern as to how the rest of the software design performs it tasks. As a result, CGF development efforts can focus on what knowledge is required and how that knowledge is invoked.

Another contribution of this research is the separation of domain-dependent issues from domain-independent issues. Domain-independent issues such as coordination and communication between CGFs can be implemented within the architecture and software design. A standardized architecture for broad classes of CGFs makes the incorporation of such issues readily available to *all* CGFs using that architecture. Ultimately, these CGFs can display more complex behaviors (e.g., combined arms tactics) than behaviors displayed by CGFs based on tightly-coupled software architectures. Perhaps more importantly, the amount of effort required to implement complicated behaviors is greatly reduced.

Finally, this research maps the unpredictability and certifiability of CGF behaviors to both skills and a psychological model. Since combat is not just a function of skills, believable CGF behaviors require the injection of additional, less tangible concepts such

as morale, aggressiveness, and intra-team support into the training environment. These concepts allow two seemingly identical CGFs to display different behaviors based on their current state as well as the state of the environment. This yields a time-limited Turing test for human actors in distributed virtual environments—given the short response time in such an environment, human actors can't distinguish between CGFs and other human actors. As a result, human actors are forced to engage CGFs as they would other human actors, reinforcing realistic tactics and increasing the overall training effectiveness.

5.2 Recommendations

There remains much that can be further explored with respect to this research effort, both in concept and in application. With the establishment of a design methodology and a supporting software architecture, most of these research areas are concentrated in the field of artificial intelligence.

5.2.1 CGF Coordination. While the DMTITE software architecture was developed with an eye towards CGF coordination and communication, no method in support of this goal was identified or implemented. This aspect of CGFs has a broad scope, necessarily addressing other concerns such as mission planning and achievement—concerns that fall within the domains of both the Long-Term and Mission-Level Decision Engines. Furthermore, a means of communicating concepts such as those identified by the combat psychology model is required to utilize the full range of behaviors supported by the model.

5.2.2 Planning. Planning is current a "hot" topic in the field of artificial intelligence and, while the software architecture's concept of a long-term decision engine implicitly supports this topic, no explicit support is provided. Real-time mission planning for CGFs remains just beyond the reach of researchers. *A priori* mission planning techniques have been identified, but fail to adequately address the dynamic nature of DVEs such as the synthetic battlespace.

5.2.3 Knowledge Acquisition. One of the more obvious assumptions of this research is that no one knowledge acquisition technique is "most suitable" for CGF develop-

ment. However, no attempt was made to validate this assumption since it was out of scope of this effort. Future research in knowledge acquisition could attempt to either validate this assumption or invalidate it by identifying an acquisition technique suitable for the broad classes of CGFs supported by the DMTITE design methodology and software architecture. Even if this assumption is validated, it may be possible to identify knowledge acquisition methods that are more applicable than others in the domain of CGF construction.

5.2.4 Verification and Validation. Another obvious assumption of this research is the behaviors of DMTITE CGFs can be verified and validated. In past efforts, this has essentially meant sitting domain experts in front of a terminal and having them say "yeah, that looks right." When CGFs display multiple skill levels, this qualitative approach works only if the domain experts can relate to the skill level being displayed. Future research could attempt to quantify the verification and validation of CGF behaviors. Such research would be extensible to other CGF development efforts as well.

5.3 Conclusions

A domain-independent design methodology and software architecture address many of the concerns raised by current CGF development efforts. By establishing a common framework from which CGFs belonging to various domains can be constructed, development (and ultimately training) costs are reduced. CGFs within a given domain can exploit commonalities shared by other CGFs within that domain. Domain-independent concepts such as combat behaviors and skills can be implemented for *all* CGFs, not just those in a specific domain. The software architecture removes the threat generation system from a single simulator and makes it globally accessing to *all* simulators within a distributed virtual environment. This allows all simulators to observe threat behaviors at a consistent fidelity level. CGFs built from a common architecture inherit the domain-independent features of that architecture. This inheritance allows CGFs to form a complex, coordinated threat environment. In addition, the domain-independent concepts of skills and combat psychology minimize exploitable patterns of CGF behavior, increasing the overall training effectiveness of the distributed virtual environment.

While this research was conducted in support of a training environment for human pilots, the resulting design methodology and software architecture are easily adaptable to other CGF development efforts. This is because neither the methodology nor the architecture are coupled to a specific domain (air, land, surface, or space). In addition, the methodology isn't bound to a specific set of knowledge acquisition, verification, or validation tools; it allows knowledge engineers to use any method they deem appropriate to their specific development effort. Nor is the software architecture bound to a specific set of physical models—software engineers may develop additional models as necessary and easily incorporate them into the overall software design. This shifts the CGF development effort away from "structure implementation" and towards "knowledge implementation."

Despite these assertions, the ultimate success of this research can be gauged by how widely accepted and used the methodology and architecture are. Applying this approach strictly within the domain of DMTITE without evaluating its fitness for other CGF development efforts will not not prove anything. Successfully implementing CGFs using this methodology and architecture in other development efforts is critical for this research to be validated.

Appendix A. Common Object Database Architecture Implementation

The CODB architecture implemented for the DMTITE project is a significant modification of previous CODB implementations. The exact implementation, as well as the ramifications of these modifications, is discussed in the following sections.

A.1 Previous CODB Implementations

There have been two previous versions of the CODB, one developed in support of the Virtual Cockpit project (1), the other in support of the Intelligent Wingman project (38). The initial version of the CODB was built using the assumption that only a single application per host would use the CODB. While this supported data sharing across a network, it greatly reduced the number of entities supported by the CODB concept. A modification was made to support multiple processes on a single host; however, these processes had to be spawned from the same parent process (38). This implementation falls short of the functionality required by the DMTITE project, in which CGFs are viewed as independent processes, not child processes of a single parent.

The two implementations also shared a set of problems making both undesirable for supporting DMTITE. Neither could support multiple CODBs on a single host; in fact, it could be demonstrated that bringing up a second CODB would cause the host to crash. Neither implementation allowed for "selective" retrieval of the contents of a container; the application either retrieved the contents of the *entire* container or nothing at all. Finally, both implementations allowed applications to directly access the CODB memory contents. This dangerous access technique not only allowed the application to access its assigned memory segment within the CODB, but any other memory segment as well. Clearly, modifications were required to bring the CODB implementation to the level of functionality support by its concept and required by DMTITE.

A.2 Extending the CODB Concept

While previous implementations of the CODB failed to meet the original concept, the original concept did not adequately address issues germane to the DMTITE effort. To

support the operational concept of the synthetic battlespace, containers must be able to distinguish between and properly handle *persistent* and *non-persistent* information. Furthermore, the original CODB concept envisioned a single type of "user"—an application concerned with the information in the CODB. However, an implicit requirement of the CODB was that it be "recursively defined": on a single host, tightly-scoped CODBs could be connected to larger CODBs and so forth, ending with a CODB containing the totality of the DVE (Figure A.1). In order to support this concept, a new "user" was required—one that viewed the information in the CODB only with regards to routing. These concerns are addressed in the DMTITE implementation of the CODB.

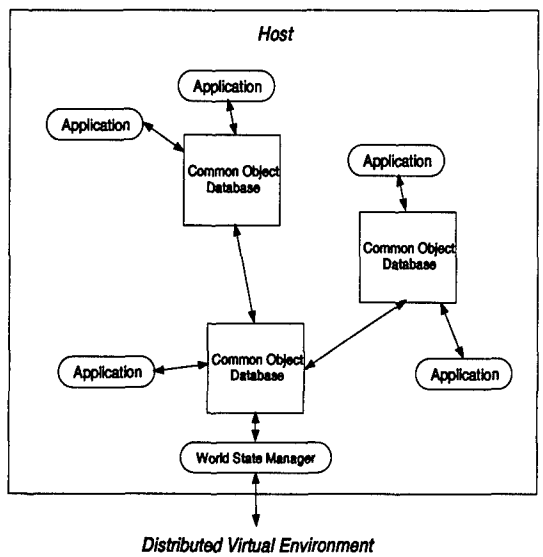

Figure A.1 "Recursively Defined" CODB System

A.2.1 Persistent and Non-Persistent Containers. There are actually two types of containers being handled by the CODB. The information in *persistent* containers is accessed one or many times by the applications connected to the CODB. Persistent containers can be updated; however, *all* information in the container remains available to the applications accessing it. On the other hand, the information in *non-persistent* containers is meant to be accessed at most once by each application; when all applications have accessed specific information, that information is removed from the container. Non-persistent containers

can be updated; however, *only that information not accessed previously* by an application is available.

This distinction is made to support the two types of information broadcast in a distributed environment. *Data* is persistent information—for example, entity state information. If an entity doesn't move, its current position remains known to other entities. Only when the entity "leaves" the distributed environment is its information removed. *Events* are non-persistent data. An entity affected by a missile detonation should process this information once and ignore any duplicate notifications of the same event. Therefore, event information should be removed once all affected entities have received it.

The DMTITE CODB implementation processes these two types of containers by monitoring the number of readers for each container as well as the identity of each reader. Non-persistent information is passed only once to each reader; however, that information remains available until all readers have accessed it. The CODB removes non-persistent information from a container only after it detects *all* readers have accessed the container. Persistent containers are monitored as well, but no action is taken when all readers have accessed these containers.

A.2.2 "Pass-Through" Applications. There are actually two views of CODBs within a "recursively defined" CODB system. A *subordinate* CODB contains a strict subset of the information in the *supervisory* CODB to which it is logically attached. In turn, a supervisory CODB can be subordinate to another, more inclusive CODB. This chain of subordinate/supervisory CODBs ends with the World State Manager's CODB; since it contains the total state of the DVE, it cannot be subordinate to any other CODB.

The original CODB is nothing more than a data repository for use by other applications and is incapable of supporting this concept. A CODB is not concerned with the type of information contained within it, nor does it initiate a connection with the source of that information. Therefore, a new type of CODB "user", an *object manager*, is required to transfer containers between CODBs in an intelligent manner.

As implemented for DMTITE, an object manager is initialized (via an initialization file) with the type of information to be passed to its subordinate CODB as well as how

often the information is to be updated. At the specified intervals, the object manager activates, passes the information matching its criteria to the subordinate CODB, and passing all new information from the subordinate CODB to the supervisory CODB. Each object manager must be terminated externally; however, object managers detect these signals and disconnect from any CODBs they access. This is necessary since each CODB releases its disk space only after *all* connections to it have been terminated.

A.3 CODB Logical View vs. CODB Implementation View

Even with the extended CODB concept, there remains a difference between how the CODB *logically* operates and how its *implementation* operates. These two views are briefly discussed below, along with the justification for the differences between the two.

A.3.1 Logical View. From a logical viewpoint, an application views the CODB as a collection of containers from which information is extracted and to which information is written (Figure A.2). The order in which information is placed within a container remains constant: an application writes its own information to its assigned slot, and can read information from another application by accessing that application's assigned slot. In this view, applications always obtain the most up-to-date information each time a container is read.

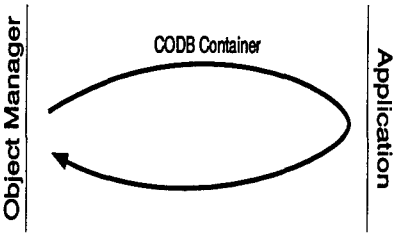

Figure A.2 Logical View of a CODB

A.3.2 Implementation View. As implemented, a CODB consists of two unidirectional halves (Figure A.3). From the viewpoint of an application attached to the CODB, one half consists of "incoming" containers—information *to be used by* the applications attached to the CODB. The other half consists of "outgoing" containers—information

generated by the applications connected to the CODB. Therefore, most applications may only read "incoming" containers and may only write "outgoing" containers. Although it is desirable to allow applications to retrieve the most up-to-date information from "outgoing" containers, the decision to disallow this was made to simplify processing within the CODB. As a result of this decision, processing time is decreased since updated ("outgoing") information can be retrieved and forwarded without scanning the entire contents of the respective container.

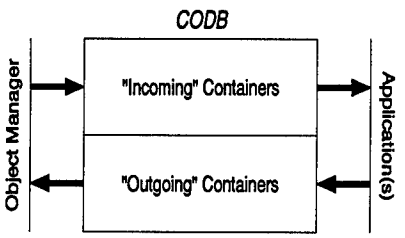

Figure A.3 Implementation View of a CODB

A.4 DMTITE CODB Object Model

The DMTITE CODB object model that supports the extended CODB concept is shown in Figure A.4. As established by Rumbaugh (26), rectangles denote classes and subclasses, while triangles are used to indicate inheritance. There are four classes within the CODB object model, as described below; one is a virtual base class, while the others represent the ways an application can "view" the CODB.

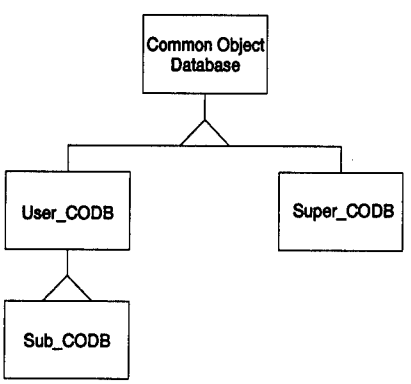

Figure A.4 DMTITE CODB Object Model

A.4.1 Common Object Database. The Common Object Database class is the virtual base class for all CODB implementations. This class defines the implementation of some member functions inherited by derived classes; however, it defines the interface for most member functions (leaving the derived classes to instantiate their own versions of these functions). Because this class forces derived classes to define their own version of some member functions, no CODB of this type can be directly instantiated.

A.4.2 User_CODB. The User_CODB is the CODB as viewed by an application concerned with the *contents* of the CODB. This CODB object assigns a single slot in each container to be written to by an application (in direct support of the original CODB concept). Unlike previous CODB implementations, applications never directly access this memory "slot". Instead, the underlying data structure within the CODB is responsible for determining where to place incoming information in the shared memory arena. Applications connected to a User_CODB may only read incoming containers and may only write outgoing containers.

A.4.3 Sub_CODB. This subclass is how the object managers view the CODB to which they are *subordinate.* This view of the CODB is similar (but not identical) to the view applications have of the User_CODB. As a result, the Sub_CODB is derived from the User_CODB class. Object managers may only read incoming messages from and write outgoing messages to the CODB to which they're subordinate. However, since object managers are responsible for passing along information from multiple applications, they're allowed to write to multiple slots within the Sub_CODB.

A.4.4 Super_CODB. This subclass is how the object managers view the CODB they *supervise.* In this view, the role of "incoming" and "outgoing" containers is reversed: the object manager reads "outgoing" containers and writes "incoming" containers. As with the Sub_CODB, object managers are allowed to multiple slots within containers, since they are passing along information from multiple (possibly distributed) applications.

A.5 CODB Implementation

The actual implementation is mapped directly from the extended CODB concept, taking the implementation viewpoint discussed in section A.3.2. IRIX shared memory arenas and Standard Template Library (STL) data structures were used to implement the interprocess communication and storage functionality aspects respectively. These changes also required the CODB container data structures to be re-engineered.

For DMTITE, the *contents* of the containers were also implemented as classes. However, this aspect of the implementation does not directly affect the CODB implementation (although it is related to it) and is not discussed here.

A.5.1 Shared Memory Arenas. Under UNIX, the size of shared memory is normally limited to 64 KB. Silicon Graphics (SGI) addresses this limitation in their IRIX operating system through the use of *shared memory arenas.* An arena is a disk file that acts as additional shared memory (28). This interprocess communication mechanism is implemented as extended functions in the C library, operating in user space without system calls. While this is convenient from a programming perspective, these arenas are IRIX-specific. As a result, this implementation of the CODB is not portable to other hardware platform or even SGI machines using another operating system. If the CODB is to be implemented on another platform, the interprocess communication aspects will need to be re-engineered.

A.5.2 Maps and Vectors: The Standard Template Library. The DMTITE CODB implementation relies heavily on the Standard Template Library (STL) *map* and *vector* classes. The STL is a collection of commonly-used data structures, defined as templates so they are usable by any data type (even user-defined). The underlying structure of the STL is based on red-black trees, guaranteeing *O(lg n)* access time in the worst case (9). In addition, STL handles memory allocation and deallocation automatically. Finally, the STL has become part of the ANSI standard for C++, which allows the storage aspects of this CODB implementation to be portable across hardware platforms.

In STL, a *map* defines a relationship between a key and its satellite data, in a manner similar to how a telephone book maps a name to a number (2). A map is similar to an array in that the satellite data can be accessed using the key as an index. When a new index is referenced, the map dynamically allocates the appropriate amount of memory for the satellite data. Each index in a map is unique; therefore, when an existing index is referenced the map overwrites the associated satellite data as appropriate. A *vector*, on the other hand, is a linear list of flexible size. For vectors, the user explicitly places information at the head or tail of the vector—no indexing is supported. In the worst case, then, the entire vector must be searched to find a specific piece of data.

In the CODB, maps are used within the shared memory arena to map slots to actual memory locations. In addition, these maps are stored in a "map of maps" which allows applications to locate and access a specific memory slot in *O(lg n)* time. To support the CODB concept of double-buffering, two such "map of maps" are instantiated: one for incoming containers, one for outgoing containers. (For additional information on double-buffering, refer to Stytz, et al. (34) or Zurita (38).) Vectors are used for lower-level data structures in which each element will most likely be accessed sequentially. Although these data structures are not discussed here, they are part of the CODB. Therefore, the concept of vectors is included here for completeness.

A.5.3 Containers. As proposed by Stytz, et al., a CODB container is divided into categories and slots, as shown in Figure A.5. The *primary category* is the coarsest division of container; an example of a primary category in DMTITE would be "red force", "blue force", "neutral forces", and "unknown forces". A primary category is further divided into *secondary categories*, which are divided into *tertiary categories*, and finally *specific categories* (the smallest possible division of a CODB container). Slots contain the actual data of the container; slot contents are provided by applications, not the CODB.

An application manipulates a container as a collection slots, each of which is a data structure shown in Figure A.6. The first six fields of this data structure map the slot to a particular container location; the last field contains the actual data. The *container ID* field indicates the container to which the data is assigned, while the *entity ID* field contains

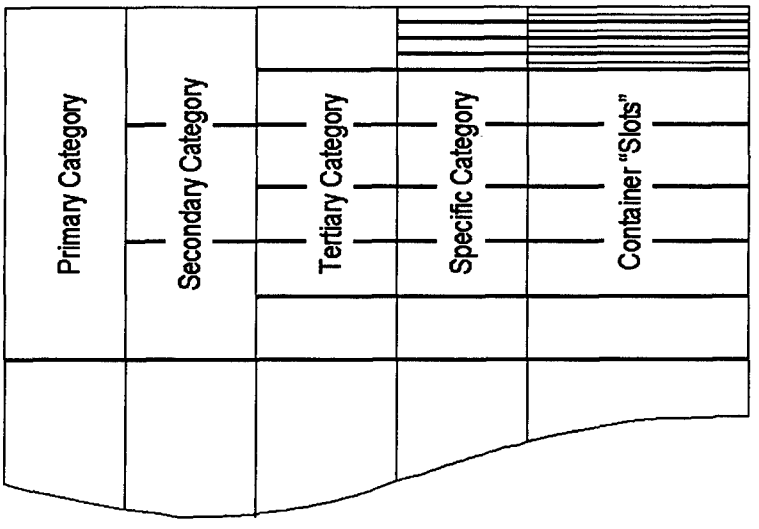

Figure A.5 CODB Container Anatomy

a simulation-unique identifier for the application to which the information pertains. This ID may be the application that originally broadcast the information to the DVE (e.g., entity state information), or the application targeted by the information (e.g., a fire or collision event). The remaining four fields are the categories relevant to the information; each category is represented as a 32-bit value, allowing a total of 32 unique identifiers per category. (The choice of four categories was strictly arbitrary; the actual number can be increased or decreased by modifying the corresponding data structure.) "Wildcard" categories are permitted, allowing a slot to correspond to any value of the category in question.

A.6 Ramifications

This CODB implementation addresses the shortcoming identified with previous implementations and brings the CODB concept fully to life for the first time. Applications can now access some or all of the contents of a container. Multiple CODBs can be hosted on a single platform, whether in support of single or multiple DVEs. Applications do not directly access the contents of the CODB; instead, copies of this information are returned, allowing the application the flexibility to modify that information with no impact to the

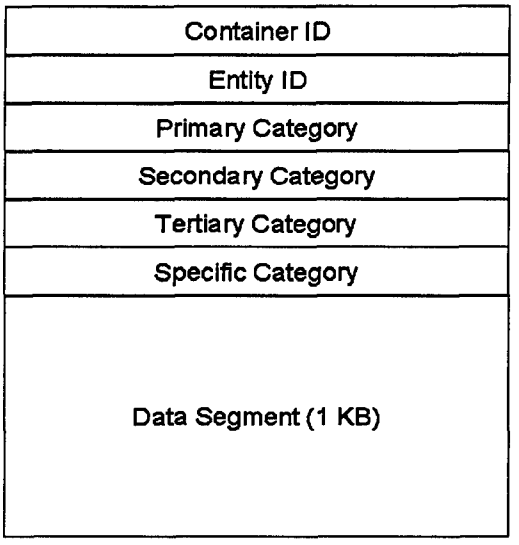

Figure A.6 CODB Container "Slot"

CODB. In addition, the original CODB concept was extended to address concerns germane to DVEs. Persistent and non-persistent containers control how information is passed to applications, while object managers pass information between two CODBs.

Perhaps the most significant ramification of this implementation is the reliance on shared memory arenas. This concept is unique to the IRIX operating system; no other version of UNIX implements these arenas. As a result, this version of the CODB performs only on Silicon Graphics platforms. If the CODB concept is to be implemented on other platforms (e.g., Solaris or Linux), another means of allocating large amounts of shared memory will need to be determined.

A.7 Conclusions

Previous versions of the CODB architecture were implemented through the use of several simplifying assumptions that rendered them ineffective for the DMTITE project. By eliminating these assumptions, the current implementation of the CODB provides a robust architecture that can be utilized by multiple applications on a single host, handles user-defined containers, and supports the "recursive" implementation envisioned by its original designers. Since the actual CODBs are implemented as shared memory arenas,

and not processes, some additional overhead is introduced through the use of "object managers". These applications route containers between CODBs at user-defined intervals. Whether these "object managers" have a significant negative impact on the DVE remains to be determined.

Appendix B. DMTITE Design Methodology

Figure B.1 illustrates the overall design methodology for DMTITE computer generated forces (CGFs). A more detailed explanation follows in section B.2.

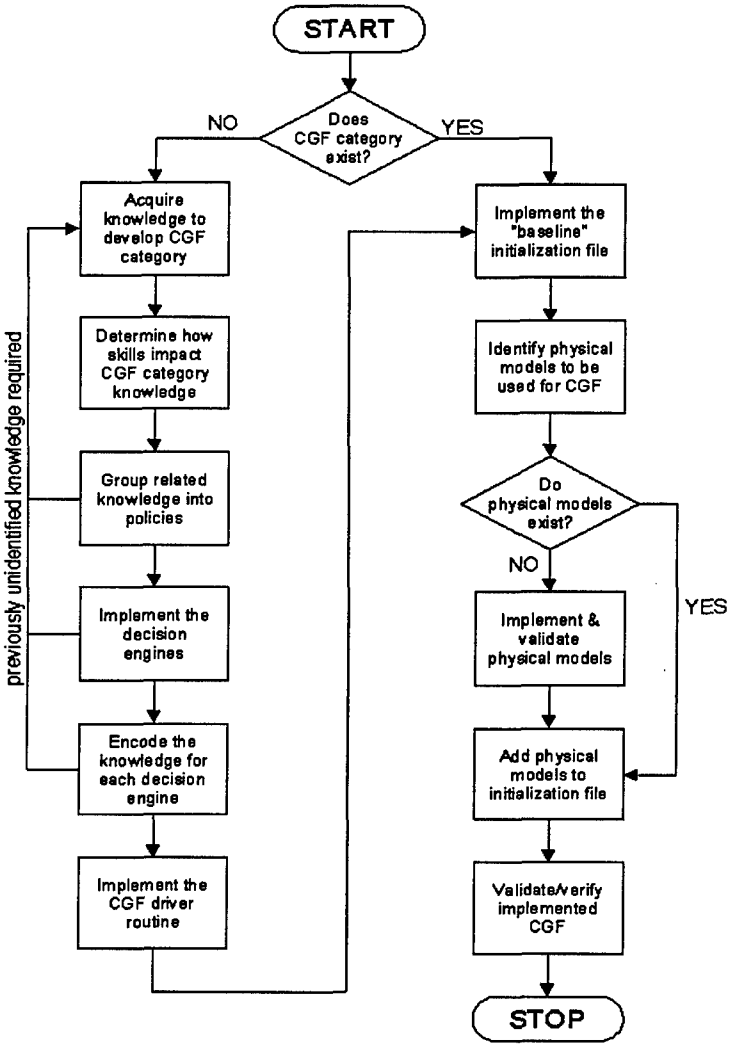

Figure B.1 DMTITE Design Methodology

The left half of the methodology represents the implementation of the cognitive model (the "Cognitive Representation Component", or CRC) for a *given CGF category.* This portion of the methodology is performed only when a CGF category has not yet

been implemented or when knowledge needed by a CGF has not yet been implemented within a CGF category. The right half of the methodology represents the implementation of the physical model (the "Physical Representation Component", or PRC) for a *specific CGF*. During this portion of the methodology, the software engineer is concerned with implementing specific physical components (e.g., weapons, sensors, and kinematic models), identifying the knowledge bases to be used by the CGF, and specifying the CGF's entity profile values. The first half of the methodology is the most time-consuming (15) but is performed less often than the second half.

B.1 Assumptions

This design methodology assumes a taxonomy has been defined for the CGFs to be supported. This taxonomy must support the concept that all CGFs within a given category share a common cognitive process, although the knowledge manipulated by this process may vary slightly between CGFs. Additionally, this methodology assumes the supporting software architecture has been previously implemented, verified, and validated. If the underlying architecture has not been verified and/or validated, it will be difficult (if not impossible) to determine if processing errors are the fault of the architecture or the knowledge being input into that architecture.

B.2 Design Methodology Process

1. *Determine if CGF category has been implemented.* This step of the methodology is concerned with scoping the amount of knowledge acquisition to be performed by the knowledge engineer. If the CGF category has not yet been implemented, the knowledge engineer must obtain *all* appropriate knowledge from domain experts, manuals, and so forth. If, on the other hand, the CGF category does exist, the knowledge engineer must determine if any *additional* knowledge is required by the CGF being implemented. If all necessary knowledge has been acquired and implemented, no knowledge acquisition is necessary and the software engineer may begin work at step 7.

2. *Acquire knowledge to develop CGF category.* During this step, the knowledge engineer elicits and documents knowledge from appropriate sources using any appropriate and available knowledge acquisition technique(s). The knowledge acquisition performed in this step is limited to that identified in step 1; however, the knowledge engineer should also identify any modifiers (e.g., skills) for the knowledge in question. Although the knowledge engineer may choose to progress to the next step only after knowledge acquisition is complete, this step (along with steps 4, 5, and 6) will most likely be part of an iterative process.

3. *Determine how skills impact the CGF category knowledge.* The knowledge engineer determines exactly how any modifiers identified in step 2 affect the knowledge just acquired. For this, a "refinement" approach is recommended: the knowledge engineer should identify how an unmodified entity would view the knowledge, then determine the impact of modification. Does the scope of the knowledge increase? Decrease? Shift one way or another? Or does it exhibit a combination of these behaviors?

   Another approach that can be used to quantify skills is *input modeling.* This technique, used extensively in traditional simulations, maps real-world data to mathematical distributions (3). The resulting equations can then be used to map skills to the corresponding value in the distribution.

   If additional knowledge is required that hasn't already been acquired, the knowledge engineer should return to step 2 before continuing.

4. *Group related knowledge into policies.* During this step, the knowledge engineer first identifies knowledge that "feeds off" other knowledge (e.g., rules that require other rules to have fired previously), combining this knowledge into tightly- coupled knowledge bases known as *policies.* As this step progresses, the knowledge engineer should ensure no knowledge is duplicated in multiple policies; if this occurs, the affected policies should be decomposed into smaller, more atomic policies.

5. *Implement the decision engines.* With the assistance of the knowledge engineer, the software engineer determines which inferencing strategy will be used for each decision engine. The software engineer also reviews the knowledge, determining what

state information each engine requires from the physical state information interface (PSII). If state messages haven't yet been implemented for this information, the software engineer designs and implements the appropriate data structures. If the CGF category hasn't been implemented previously, the software engineer must implement and test each of the decision engines. If the CGF category has been implemented previously, the software engineer must ensure the required state information is properly extracted from the PSII.

If either the knowledge engineer or the software engineer determine additional knowledge not already acquire is required, the knowledge engineer should return to step 2 before development continues.

6. *Encode the knowledge for each decision engine.* The software engineer encodes each policy into a format readable by the decision engine(s) accessing it. Each policy is maintained in a separate file; Appendix C contains the current formats of policy files.

   Again, if during this step additional knowledge is required, the knowledge engineer should return to step 2 before further development occurs.

7. *Implement the "baseline" initialization file.* The "baseline" initialization file contains initialization information common to all CGFs in a given category (Appendix E contains the format of these files.) Among other things, this file specifies the inferencing strategy and knowledge bases used by each decision engine; the variables that define the category's entity profile; the policies that comprise each knowledge base; and the files that define the policies. If the CGF category hasn't been implemented previously, the software engineer must develop the "baseline" file. If the category does exist, the software engineer must ensure any new state information, entity profile variables, and/or policies are specified in the initialization file.

   From this "baseline" initialization file, the software engineer derives the specific initialization file for the CGF being developed, although at this step the file is incomplete. The knowledge engineer specifies the values (between 0.0 and 1.0 inclusive) for each of the entity profile variables during this step, although these may be changed

as necessary. Data required by each of the decision engines (e.g., weighting values for case-based engines) is also specified at this time.

8. *Identify the physical models to be used by the specific CGF.* Both the knowledge engineer and software engineer identify the physical models to be used by the CGF. These models provide the state information for the cognitive representation, and are directly or indirectly identified during the knowledge acquisition phase (step 2). This step consists of mapping state information to inputs; if no corresponding input exists, the CGF must be able to deal with this uncertainty by using the knowledge available to it. The knowledge and software engineers also decide the level of fidelity required by the CGF during this step.

   If all identified physical models have been implemented previously, development continues at step 10.

9. *Implement and validate the physical models.* Each of the missing physical models must be implemented by the software engineer. This step consists of identifying the functionality to be implemented, determining the state information to be retrieved from the sensor interface, and ensuring the corresponding state information is stored in the physical state information interface. The fidelity of the model must correspond to the level of fidelity identified in step 8. The software engineer also performs sufficient testing of the operation of each physical model developed during this step.

   The software engineer may also need to add appropriate code to the model initialization method of the physical representation to allow each newly designed model to be instantiated at runtime.

10. *Add physical models to the initialization file.* Each physical model to be used by the CGF is added to the initialization file by the software engineer. Any arguments required by each physical model (e.g., rounds of ammunition, pounds of fuel) are also specified during this step. The physical models must be specified in the order they are processed during runtime; models requiring inputs from other models should appear *before* the models providing the inputs.

11. Validate and verify the implemented CGF. Both the knowledge engineer and software engineer validate and verify the CGF by observing its behaviors in the virtual battlespace. For completeness, the entity profile values should be changed during this step to ensure the CGF displays an appropriate range of behaviors. Newly developed physical models and decision engines should be examined closely to ensure they function as expected. The software engineer should use validated software engineering techniques to validate the CGF's operation; the knowledge engineer should use accepted verification and validation techniques to evaluate the CGF's behaviors.

Appendix C. Policy File Formats and Examples

Policies are self-contained knowledge bases that form the "building blocks" for the knowledge bases accessible by DMTITE decision engines. Each policy and the knowledge base(s) that access it is specified in the initialization file (see Appendix E). The actual knowledge contained in a policy is specified in another file called a *policy file.* Although knowledge used by the three inferencing strategies supported by DMTITE is derived from a common representation, each requires a specialized policy file. The expected formats of these files, as well as a few simple examples, are described in the following sections.

C.1 General Expressions

Each knowledge representation is comprised of one (or more) expressions. These expressions define preconditions (conditions which must be satisfied for a knowledge representation to "fire"), true postconditions (the set of facts inferred if a knowledge representation's preconditions hold), or false postconditions (the set of facts inferred if a knowledge representation's preconditions are false). A knowledge representation can have one, none, or several precondition and/or postconditions.

In general, expressions consist of two variables and a relationship operator (Figure C.1). The variable on the left hand side of the expression *never* has a value explicitly defined in an expression, while the variable on the right hand side *may* have a value explicitly defined. Variable names can be any sequence of alphanumeric characters (no spaces) desired, with `CONSTANT` understood by DMTITE to represent a constant value. Variable types currently supported by DMTITE are shown in Table E.1 (Appendix E). Supported relational operators are similar to those used in the C/C++ programming languages, as shown in Table C.1.

Figure C.1 General Expression Format

Table C.1 Relational Operators Supported by DMTITE

Relational Operator	*Symbol*
equal to	`==`
not equal to	`!=`
less than	`<`
less than or equal to	`<=`
greater than	`>`
greater than or equal to	`>=`
assignment	`=`
retract	`retract`

The format of a `retract` expression is different from other expressions. These expressions only have a single variable, specified on the left hand side of the `retract` operator. When a `retract` expression is executed, it removes the specified variable from memory.

Usually, the value specified for a variable must match that variable's type. However, DMTITE currently supports three "value words" for readability. `true` and `false` are used to define the corresponding boolean values. `any_value` can be used by any variable type, and is used to define "wildcards". In other words, a variable assigned `any_value` in an expression will always be evaluated as true, regardless of the value that variable is actually assigned in memory. `any_value` was implemented to support case-based reasoning (see section C.3). Finally, variables can be assigned formula values. These are expressed in postfix notation (e.g., "2 2 +" to add two and two together) and may reference other variables. The value of a formula is calculated each time the corresponding expression is evaluated.

C.2 Rule-Based Policies

The rule-based inferencing strategy used in DMTITE is based on the standard *if-then-else* format. The *if* portion of the rule represents the preconditions, the *then* portion represents the true postconditions, and the *else* portion represents the false postconditions. A rule in DMTITE consists of zero, one, or several precondition, false postcondition, and true postcondition expressions (see the previous section). The current rule-based

inferencing engine in DMTITE is very basic; however, work continues on integrating a public-domain inferencing engine known as CLIPS (C Language Implementation Production System) into the DMTITE system design.

The rule-based policy file format is shown in Figure C.2. The <*Description*> section contains a single-line text description of the policy. The <*Inferencing Strategy*> section contains a single keyword defining the inferencing strategy the policy supports; for rule-based policies, the keyword is `rule_based`. Finally, the <*Knowledge Representations*> section defines the actual rules. The first line of this section defines the number of rules comprising the policy, while the remaining lines describe the actual rules. Each rule is defined by several lines. The first contains the rule's name, the number of preconditions, false postconditions, and true postconditions. The second line of each rule contains a text description of the rule. This line is followed by the expressions that define the preconditions, then the expressions that define the false postconditions, and finally the expressions that define the true postconditions. This pattern is repeated for each rule in the policy.

A portion of an existing rule-based policy is shown in Figure C.2. This particular policy defines the computable combat psychology model (see Appendix D); the two rules shown represent the conditions necessary to have a CGF hesitate or refuse to respond to orders under combat situations. These rules return boolean "flags" to the arbitration engine, which then acts upon them appropriately. Both rules shown retract the appropriate "flags" from memory when the preconditions do not hold.

C.3 Case-Based Policies

The case-based inferencing strategy used in DMTITE associates a knowledge representation to a group of facts through the use of *frames*. Frames were originally developed by Marvin Minsky, and attempt to simulate a human's ability to deal with new situations by using existing knowledge of previous events, concepts, and situations (15). Standard case-based reasoning selects a frame based on some "goodness-of-fit" functions, then modifies that frame's output to match differences between the selected frame and the current situation. The current case-based inferencing engine used in DMTITE does not modify the output of the frame it selects.

```
<Description>
text_description

<Inferencing Strategy>
rule_based

<Knowledge Representations>
#_of_knowledge_representations
representation_name #_of_preconditions #_of_false_postconditions #_of_true_postconditions
representation_description
precondition_expression_1
...
precondition_expression_r
false_postcondition_expression_1
...
false_postcondition_expression_s
true_postcondition_expression_1
...
true_postcondition_expression_t
representation_name #_of_preconditions #_of_false_postconditions #_of_true_postconditions
representation_description
precondition_1
...
precondition_x
false_postcondition_expression_1
...
false_postcondition_expression_y
true_postcondition_expression_1
...
true_postcondition_expression_z
```

Figure C.2 Rule-Based Policy File Format

Case-based policies are similar to rule-based policies, with one exception: each knowledge unit has the same number of pre- and postconditions. The expected format for a case-based policy file is identical to that for a rule-based policy (see Figure C.2), except that the *<Inferencing Strategy>* section specifies `case_based` as opposed to `rule_based`. A portion of a case-based policy is shown in Figure C.3. This particular policy is used by an anti-aircraft artillery (AAA) CGF while in search mode; the frame shown transitions the AAA CGF from search mode to target acquisition. If additional knowledge representations were added to this file, they would be required to have twelve preconditions (referencing the variables shown in the example), no false postconditions, and one true postcondition.

```
<Description>
Defines the computable combat psychology model.

<Inferencing Strategy>
rule_based

<Knowledge Representations>
2
Delayed_Response 2 1 1
CGF is anxious but not aggressive, so delay response 1 to 4 seconds.
anxiety double      >  CONSTANT double 0.70
anger double        <  CONSTANT double 0.50
delay_response retract
delay_response bool =  CONSTANT bool true
Refuse_To_Respond 2 1 2
CGF is anxious and not receptive to orders, so refuse to respond to orders.
anxiety double      >  CONSTANT double 0.85
acquiescence double <  CONSTANT double 0.80
refuse_to_respond retract
delay_response retract
refuse_to_respond bool =  CONSTANT bool true
```

Figure C.3 Portion of Sample Rule-Based Policy

C.4 Fuzzy Logic Policies

Fuzzy logic requires extensions to the general expressions not required by either case-based or rule-based reasoning. Some of the variables take the form of *fuzzy sets* which encode imprecision by mapping a range of values to a membership function. A membership function is usually nothing more than a mathematical function; DMTITE currently supports increasing linear, decreasing linear, increasing S-curve, decreasing S-curve, pi curve, weighted beta curve, gaussian curve, proportional, arbitrary, and singleton membership functions. A value's membership in a fuzzy set can be modified through the use of *hedges.* Hedges are domain-independent mathematical functions that approximate the linguistic characteristics of the hedge; the hedges currently supported by DMTITE, along with their meaning, are shown in Table C.2. When combined, these concepts support rules such as "if temperature is somewhat hot then motor speed is usually moderate." This rule can be invoked using imprecise ("fuzzy") values for "motor speed" and "temperature";

```
<Description>
Defines the cases of detecting enemy aircraft using eyes or radar/IFF.

<Inferencing Strategy>
case_based

<Knowledge Representations>
1
Target_In_Range_To_Acquire 12 0 1
Target detected and in range, ready to acquire.
current_state int         == CONSTANT int 0
search_mode int           == CONSTANT int any_value
fire_mode int             == CONSTANT int any_value
target_visible bool       == CONSTANT bool false
visual_confidence double >= visual_threshold double
accoustic_detection bool == CONSTANT bool any_value
remote_detection bool     == CONSTANT bool any_value
in_radar_LOS bool         == CONSTANT bool any_value
in_optical_LOS bool       == CONSTANT bool any_value
in_engagement_range bool == CONSTANT bool any_value
target_inbound bool       == CONSTANT bool true
time_not_visible double   == CONSTANT double any_value
next_state int            == CONSTANT int 2
```

Figure C.4 Portion of Sample Case-Based Policy

an exact motor speed can be obtained by "defuzzifying" the result of the rule. Specific information on fuzzy sets, membership functions, hedges, and defuzzification can be found in Cox (10).

The fuzzy logic policy file format is shown in Figure C.4. This format is similar to that used by rule-based and case-based policies; however, it adds a <Fuzzy Sets> section to define the fuzzy sets to be used within a policy. Fuzzy sets are defined by their type, an "alpha cut" (a minimum membership value—values below this are treated as zero), and a list of fuzzy set parameters. Each type of fuzzy set requires a different set of parameters; these parameters are defined in Cox (10) and not repeated here.

Changes also appear in the <Knowledge Representations> section of a fuzzy logic policy. Since fuzzy sets don't correspond to other data types, no variable type or value is given for fuzzy sets used in an expression. This does not exclude traditional variables from

Table C.2 Fuzzy Set Hedges Supported by DMTITE (10)

Hedge	*Meaning*
usually	contrast diffusion
above, more than	restrict a fuzzy region
almost, definitely, positively	contrast intensification
below, less than	restrict a fuzzy region
generally, usually	contrast diffusion
not	negation or complement
quite, rather, somewhat	dilute a fuzzy region
very, extremely	intensify a fuzzy region
about, around, near, roughly	approximate a scalar
vicinity of	approximate broadly
neighboring, close to	approximate narrowly

being combined with fuzzy sets; if so, variables must state their type and (if appropriate) their value.

A pricing policy example from Cox (10) was used to verify the operation of the fuzzy logic inferencing engine developed for use in DMTITE; this policy is shown in Figure C.4. While not related to the DMTITE domain, this example shows how facts can be stated using general expressions (no preconditions given) and demonstrates the general format of a fuzzy logic policy.

```
<Description>
text_description

<Inferencing Strategy>
fuzzy_logic

<Fuzzy Sets>
#_of_fuzzy_sets
fuzzy_set_name
fuzzy_set_type alpha_cut
fuzzy_set_parameters
hedges_to_apply

<Knowledge Representations>
#_of_knowledge_representations
representation_name #_of_preconditions #_of_false_postconditions #_of_true_postconditions
representation_description
precondition_expression_1
...
precondition_expression_r
false_postcondition_expression_1
...
false_postcondition_expression_s
true_postcondition_expression_1
...
true_postcondition_expression_t
representation_name #_of_preconditions #_of_false_postconditions #_of_true_postconditions
representation_description
precondition_1
...
precondition_x
false_postcondition_expression_1
...
false_postcondition_expression_y
true_postcondition_expression_1
...
true_postcondition_expression_z
```

Figure C.5 Fuzzy Logic Policy File Format

```
<Description>
Defines the example price strategy policy.

<Inferencing Strategy>
fuzzy_logic

<Fuzzy Sets>
5
HIGH
INCREASING_LINEAR 0
16.0 36.0 16.0 36.0
LOW
DECREASING_LINEAR 0
16.0 36.0 16.0 36.0
Around_Twice_Manufacturing_Costs
PI_CURVE 0
16.0 36.0 24.0 6.0
NVHigh
INCREASING_LINEAR 2
16.0 36.0 16.0 36.0
VERY NOT
Around_Competition_Price
PI_CURVE 0
16.0 36.0 28.0 4.0

<Knowledge Representations>
2
First_Knowledge 0 0 3
Builds the base price fuzzy output set.
Price                   =       HIGH
Price                   =       LOW
Price                   =       Around_Twice_Manufacturing_Costs
Second_Knowledge 1 0 1
Builds the variable part of the price output set.
Competition_Price       ==      NVHigh
Price                   =       Around_Competition_Price
```

Figure C.6 Portion of Sample Fuzzy Logic Policy

Appendix D. Computable Combat Psychology Model

The following model was based on a study of the behavioral effects of combat stress by Dr. Steven Silver, currently an associate clinical professor, Department of Psychiatry, Temple University Medical School. It was originally intended for use in Atomic Games' *Close Combat*, a computer game simulation of the Normandy campaign of World War II. According to Dr. Silver, Atomic Games elected to incoporate only the basic concept of the model in their final product, citing the limited computational power of the personal computers being targeted for the game (30).

This model was developed using literature analyzing human reactions in combat (16, 23) and the "trait-state" field of psychology. The basic concept is that each individual has an unique psychological profile, resulting in probabilistic responses in various stressful situations. The goal of the model is to be able to simulate a wide range of combat behaviors, from the rage reaction of a friend's death to a soldier who would not only disobey but kill his commander (31). When approached about incorporating his unpublished model into DMTITE, Dr. Silver readily agreed, providing the model description that follows (29).

The following sections describe trait-state psychology, the model as it was originally proposed, and how the model was actually implemented in the DMTITE project.

D.1 "Trait-State" Psychology

Trait-state psychology views human functioning in terms of fundamental traits; all people have all traits in varying degrees, depending upon genetics and life experience. This particular approach to psychology was selected because it can be translated into a numerical format, easing the programming task.

In trait-state psychology, a trait is relatively stable and doesn't fluctuate much; it represents the "normal" value for a particular individual. States, on the other hand, are derived from traits and are situational; they represent an individual's transitory response to a situation. For example, consider anxiety as a trait and as a state. A "normal" person may have an anxiety *trait* of 10 (on a 0-100 scale). When surprised by a balloon popping, a person may experience an anxiety *state* of 20. Traits and states are combined in a

situation, then behaviors are induced based on real world observations. Returning to the anxiety example, assume that an anxiety level (trait + state) of more than 70 has been observed to affect behavioral functioning. An anxious individual (with an anxiety trait of 60) would display these effects when surprised by a balloon popping (since 60 + 20 is more than the threshold value of 70). A probabilistic or other appropriate approach can be taken to induce the appropriate effects based on the situation.

In short, then, the trait-state approach to psychology evaluates a given situation by determining the current state, applying it to the existing trait, and displaying the resulting reactions and behavior of the individual.

D.2 Limitations and Assumptions

While the trait-state approach allows a straightforward approach to modeling psychology, the model is inherently incomplete. As a whole, human functioning is not computable and while group behavior is generally predictable given certain conditions, individual behavior is far less so. This model assumes the best predictor of individual human behavior is that individual's past behavior. As a result, this model is essentially stable; that is, wide variations in an individual's behavior should be uncommon, but not impossible. Furthermore, group behavior is assumed to be a compilation of individual behavior. The results of both individual and group behavior are assumed to be reflected in performance, and the results of that performance should feedback into individual and group states and traits.

This model is limited to functions relating to combat, including leadership, command, communication, control, morale, and military skills performance. Within these functions, this model allows form inhibitions and enhancements of performance, such as speed of movement, accuracy, and cooperation. The model also provides for, to a limited extent, unusual individual behavior, including running amok, charging the enemy, hesistation, and panic-stricken rout. Since the full range of potential human behavior cannot be modeled, only those behaviors most relevant to performance for will be accounted for. This model was developed utilizing the behavior of 20th century infantry; despite this, it is assumed to

be applicable to all forms of individual and team combat. This assumption seems justifiable because human psychology has remained relatively constant since it was first documented.

D.3 Definitions

There are three types of variables used in this combat psychology model. As stated previously, traits remain relatively constant; however, they serve as the initial corresponding state value, which are dynamic in nature. In addition, a series of computed variables can be derived from both traits and states. These variables are defined in the following sections.

D.3.1 Traits and States. The following explicit traits and states are used to model human behavior in combat situations. Each is assumed to be present to some degree in each individual.

1. *Stability.* A generic term encompassing emotional stability functions as opposed to particular emotions. It serves as the "governor" of emotional expression, particularly extreme emotional expression such as panic.
2. *Anxiety.* Inherent fearfulness.
3. *Anger.* A generic term encompassing the emotion of anger, this variable also accounts for aggressiveness.
4. *Humor.* More than a simple sense of humor, this variable also accounts for emotional "bounce-back" and the ability to recover from sudden shocks, losses, and other negative impactors on morale.
5. *Acquiescence.* The willingness to follow commands, orders, and other leaders.
6. *Independence.* The ability to function independently, without leadership.
7. *Charisma.* A variable reflecting aspects of personality that others find attractive.
8. *Knowledge.* This term was selected to replace "Intelligence", which has a particular meaning in military terms. It refers to military knowledge, ranging from weapons and equipment to tactics.

D.3.2 Computed Variables. The following variables are computed from the traits and states defined in section D.3.1. Note that Support, Leadership, and Morale are calculated states, not just variables.

1. *Situational Stress.* Calculated as the ratio of friendly to enemy combatants in the engagement, with addition/subtraction provided by friendly/enemy supporting fire and Fatigue.
2. *Support.* Reflects an individual's ability to psychologically support other members of the team. It is calculated from the individual's Stability, Humor, and Acquiescence values.
3. *Group Support.* Calculated as the group average of each individual's Support value.
4. *Leadership.* The ability of an individual to command the obedience of others. It is calculated from the individual's Independence, Charisma, Anger, and Knowledge values.
5. *Morale.* Calculated from an individual's Stability, Anxiety, Anger, and Humor values; the related Group Support and Situational Stress values; and the group leader's Leadership value.

D.4 Model Functionality

This section describes the model functionality prior to, at the start of, and at the conclusion of a given simulation. Functionality during a simulation is described in sections D.5, D.6, and D.7.

D.4.1 Initial Phase. The explicit traits defined in section D.3.1 are determined prior to the start of a given simulation. These values can be specified or generated randomly; if generated randomly, the values should be between 0.2 and 0.8 (on a 0.0-1.0 scale), normally distributed as shown in Table D.1.

These values are calculated once for each individual; once calculated, they represent the "baseline" values (traits) for that individual.

Table D.1 Initial Trait Value Probabilities

Value	*Distribution*
0.3	13%
0.4	20%
0.5	34%
0.6	20%
0.7	13%

D.4.2 Initial Phase Computations. Once the initial traits have been determined, the calculated variables and states are generated as follows:

$$Morale = \frac{Stability + Anxiety + \frac{Anger}{2}}{3.5} \tag{D.1}$$

Note that the Situation Stress, Group Support, and group leader's Leadership values are not included in this initial computation.

$$Support = \frac{Stability + Humor + Acquiescence}{3}$$

$$Leadership = \frac{Independence + Charisma + Knowledge + Stability + Morale}{5}$$

These values are calculated once for each individual; once calculated, they represent the "baseline" values (traits) for that individual.

D.4.3 Operational Phase Adjustments. At the start of each simulation in which an individual participates, the state Morale value is calculated as follows:

$$\text{state } Morale = \frac{\text{trait } Morale + \frac{\text{leader's } Leadership + Group\ Support}{2}}{2} \tag{D.2}$$

In addition, each of the individual's "baseline" traits become the individual's initial states for the simulation.

D.4.4 Trait Adjustments. At the end of each simulation in which an individual participates, the initial trait values are modified as follows:

$$\text{(initial trait} - \text{ending state)} \geq 0.20 \Rightarrow \text{initial trait} = \text{initial trait} + 0.05$$

$$\text{(initial trait} - \text{ending state)} \leq -0.20 \Rightarrow \text{initial trait} = \text{initial trait} - 0.05$$

D.5 State Changes

An individual's states will change in response to the situational stress of the environment and the group. In turn, the changes in the individual will induce changes in the group. To avoid perpetual loops, the sequence of evaluation should be environment, then the group. The following is an example of environmental variables and their impact (as approximated from various studies done on the psychology of the battlefield).

D.5.1 Battlefield Variables. The battlefield variables affecting individual states are shown in Table D.2; those affecting group states are shown in Table D.3. These values are fairly rigid; if preferred, a situational stress computation could be used for proportional point allocation.

D.5.2 Stress Reduction Variables. The stress reduction variables specify events that reduce or improve states both immediately (Table D.4) and over time (Table D.5).

D.6 Effects of States on Performance

Once the current state of a CGF has been evaluated, the effects of that state need to be reflected in its behaviors. There are many different ways to modify a CGF's behavior based on its state, as discussed in the following sections.

Table D.2 Battlefield Variables (Individual Elements)

Event	*Stab.*	*Anx.*	*Ang.*	*Hum.*	*Acq.*	*Ind.*	*Char.*	*Know.*
New team member	-0.05	0.05		-0.05	-0.05			
Nighttime conditions		0.10	-0.01	-0.02	0.02	-0.02	-0.01	
Reduced visibility		0.05	-0.01	-0.01	0.01	-0.01		
Indirect fire (intermittent)		0.01	0.01					
Indirect fire (continuous)[a]		0.03	0.02	-0.01	-0.01			
Sniper fire		0.02	0.01		-0.01			0.01
Light (ineffective) fire	0.05	-0.01	-0.01	-0.02			0.01	
Moderate fire	-0.03	0.08	-0.02	-0.02	-0.02		-0.01	0.01
Heavy fire	-0.05	0.12	-0.04	-0.10	-0.05	-0.01	-0.01	0.01
Ambushed	-0.03	0.10[b]	-0.02	-0.20	-0.04		-0.03	0.01
Minefield	-0.02	0.05	0.01	-0.01	-0.03		-0.02	0.01
Attacked by inferior force		0.05	0.08		0.01		0.01	0.01
Attacked by force of equal size		0.06	0,02	-0.01				0.01
Attacked by superior force	-0.01	0.06	-0.01	-0.02	-0.01		-0.01	0.01
Attacked by overwhelming force		0.10[c]	-0.01	-0.10	-0.04	-0.02	-0.02	0.01
Ambushing an inferior force		0.02	0.10	0.02	0.02		0.01	0,01
Ambushing force of equal size		0.03	0.10		0.01			0.01
Ambushing a superior force		0.04	0.10	-0.01			-0.01	0.01
Supporting fire on call			0.10	0.02	0.05	0.04	0.02	
Close quarters combat	0.01	-0.02	0.01	0.01	0.01	0.01		
Encountering dead enemy	-0.01	0.02	0.01	0.01	0.01			0.01
Encountering wounded enemy	-0.01	0.01	0.03					0.01

[a]Every 15 minutes
[b]Every 15 minutes
[c]Every 30 minutes

Table D.3 Battlefield Variables (Group Elements)

Event	*Stab.*	*Anx.*	*Ang.*	*Hum.*	*Acq.*	*Ind.*	*Char.*	*Know.*
Team member wounded (TCR[a] $\leq$ 10%)	-0.02	0.02	0.02	-0.01				
Team member wounded (10% < TCR < 40%)	-0.03	0.04	0.04	-0.02	-0.01	-0.01		
Team member wounded (TCR $\geq$ 40%)	-0.04	0.05	0.04	-0.05	-0.02	-0.02		
Team member killed (TCR $\leq$ 10%)	-0.04	0.04	0.04	-0.02		-0.01		
Team member killed (10% < TCR < 40%)	-0.05	0.05	0.05	-0.05	-0.02	-0.02		
Team member killed (TCR $\geq$ 40%)	-0.06	0.07	0.05	-0.10	-0.03	-0.03		
Team leader wounded	-0.04	0.05	0.03	-0.05	-0.03			
Team leader killed	-0.08	0.10	0.03	-0.20	-0.05	0.02		
Incorrect orders given	-0.03	0.05	0.05	-0.02	-0.09	0.01		

[a] *Team Casualty Rate*; percentage of team wounded or killed

D.6.1 Tasks Relating to Reaction Time. Tasks relating to reaction time, such as taking cover, are affected as follows. *Delay* is a randomized variable between 0 and 4 seconds.

$$(Anxiety > 0.70) \wedge (Anger < 0.50) \Rightarrow Delay$$

$$(Anxiety > 0.85) \wedge (Acquiescence < 0.80) \Rightarrow Delay + 10 \text{ seconds}$$

D.6.2 Accuracy and Weapons Handling Effectiveness. In general, effectiveness is reduced when Knowledge is less than 0.50 and is enhanced when Knowledge is greater than 0.80 (29). To model the effects of fear on accuracy, rate of fire, and other related tasks (23), use Morale as follows.

Table D.4 Stress Reduction Variables (Immediate Effects)

Event	*Stab.*	*Anx.*[a]	*Ang.*	*Hum.*	*Acq.*	*Ind.*	*Char.*	*Know.*
Issued more effective equipment		-0.05						
Issued new clothing				0.05				
Successful defense		-0.05	0.01	0.05	0.03		0.01	0.01
Successful attack	-0.02	-0.06	0.04	0.05	0.04		0.01	0.01
Eating a meal	0.01	-0.10	-0.01	0.02	0.01		0.01	

[a]If value > 0.70, effects are doubled

Table D.5 Stress Reduction Variables ("Timed" Effects)

Event	*Stab.*	*Anx.*[a]	*Ang.*	*Hum.*	*Acq.*	*Ind.*	*Char.*	*Know.*
Sleep[b]	0.01	-0.03	-0.02	0.01	0.01			
No fire, secure position[c]	0.01	-0.03	-0.02	0.01	0.01			
No fire[d]	0.01	-0.02	-0.01	0.01	0.01			

[a]If value > 0.70, effects are doubled
[b]Every 30 minutes
[c]Every 30 minutes
[d]Every 30 minutes

$$Morale > 0.80 \Rightarrow \text{increased effectiveness}$$

$$Morale < 0.50 \Rightarrow \text{reduced effectiveness}$$

To account for the dominant nature of fear on the battlefield, an additional degradation in effectiveness occurs when Anxiety is greater than 0.80.

D.6.3 Obedience to Orders. This effect reflects whether or not an individual will obey orders, not how quickly those orders will be carried out. Under "normal" conditions, the orders will be acted upon immediately; however, the individual can also *Hesistate* (fail to carry out the orders until they are repeated) or *Disobey* (fail to carry out the orders

regardless of how many times they are repeated). The latter two conditions are defined as follows:

$$(Morale < 0.40) \wedge (Anxiety > 0.70) \wedge (Acquiescence < 0.40) \Rightarrow Hesistate$$

$$(Morale < 0.40) \wedge (Anxiety > 0.80) \wedge (Acquiescence < 0.35) \wedge$$
$$(Support < 0.50) \wedge (\text{leader's } Leadership < 0.50) \wedge (Random > 0.50) \Rightarrow Disobey$$

(*Random* is a randomized value between 0.0 and 1.0 inclusive.) There are also extremely rare circumstances where an individual will not only disobey an order but also strike out at the issuing authority. In the Vietnam conflict, this was known as "fragging" and can be defined as follows:

$$(Morale < 0.40) \wedge (Anxiety > 0.80) \wedge (Acquiescence < 0.30) \wedge$$
$$(Support < 0.40) \wedge (\text{leader's } Leadership < 0.50) \wedge (Random > 0.70) \Rightarrow Frag$$

D.7 Unusual Behaviors Induced by States

A CGF's state not only affects its behaviors, but may also induce "unusual" behaviors for given situations. These behaviors, and the states that induce them, are discussed in the following sections.

D.7.1 Panic Reactions. Most studies emphasize the role of isolation when an individual flees the battlefield by breaking and running. Conversely, the greatest preventative appears to be peer support followed closely by the individual's perception of team leadership. If one individual breaks, the probability of other team members breaking increases as well.

$$(Morale < 0.50) \wedge (Anxiety > 0.80) \wedge (Stability < 0.50) \wedge$$
$$(Support < 0.40) \wedge (\text{leader's } Leadership < 0.40) \wedge (Random > 0.50) \Rightarrow Flee$$

D.7.2 Heroism. This type of behavior has not been very well studied (16), as most armed forces are not so much concerned with understanding the behavior of those rare people called "heroes" as they have been in trying to prevent the far more common behavior of individuals overcome by anxiety. Despite this, *Heroism* can be simulated when the following combinations of states exists:

$$(Support > 0.85) \wedge (Morale > 0.60) \wedge (Anger > 0.70) \wedge$$
$$(Independence > 0.75) \wedge (Random > 0.50) \Rightarrow Heroism$$

D.7.3 Atrocities. Atrocities are displayed in two ways: killing unarmed civilians and killing disarmed and taken prisoners-of-war. As with heroism, this behavior has not been well studied (16). However, this behavior can occur when the following conditions hold:

$$(Morale < 0.50) \wedge (Anger > 0.80) \wedge (\text{leader's } Leadership < 0.50) \wedge$$
$$(Random > 0.70) \Rightarrow Atrocity$$

D.8 Incorporating the Model into DMTITE

The traits and states aspects of this model have been combined with the "skills vector" identified by Santos. et al., for their general CGF architecture (27). The initial values are explicitly defined by the user in the CGF initialization file; they are not randomly generated. The entire model is maintained by the Arbitration Engine; this engine is responsible for updating the states of the entity and ultimately displaying the behaviors

corresponding to the current state of the CGF. The current implementation of the model is expressed as a set of rules; therefore, the Arbitration Engine uses rule-based inferencing to determine the state of a given CGF.

The group aspects of this model have yet to be incorporated into DMTITE. While the software architecture is designed to allow the inclusion of cooperation and coordination between CGFs, the actual means to do so have yet to be determined. As a result, CGFs have no means of "sharing" information (such as their traits and states, or their behaviors). Unfortunately, the group aspect is a large portion of Dr. Silver's model and many of the behaviors supported by the model cannot be currently displayed by DMTITE CGFs.

Appendix E. Initialization File Format and Example

A DMTITE CGF is defined, in part, by the file used to initialize it. This appendix defines the expected format of the CGF initialization file. Each entry in the initialization file is presented and defined values for specific entries are also identified. A partial example of a working initialization file concludes this appendix.

E.1 Initialization File Format

The expected format for a DMTITE initialization is shown in Figure E.1. The initialization file is divided into several sections, each of which is described below.

```
<Entity Information>
callsign
nationality
domain
category
specific

<Initial Configuration>
WGS_84_coordinates      0.0    0.0    0.0
euler_angles            0.0    0.0    0.0
linear_velocity         0.0    0.0    0.0
linear_acceleration     0.0    0.0    0.0

<Environment Information>
exercise_ID
entity_ID
lead_entity_ID
connect_to_CODB         name_of_CODB
domains_of_interest     # (land surface air space)

<Entity Profile>
#
stability          0.0
anxiety            0.0
anger              0.0
humor              0.0
acquiescence       0.0
independence       0.0
charisma           0.0
knowledge          0.0
skill_name         0.0

<Policies>
#
policy_name policy_filename
```

```
<Knowledge Bases>
#
knowledge_base_name inferencing_strategy #_of_policies
knowledge_base_description
policy_name(s)

<Long-Term Engine>
inferencing_strategy #_of_knowledge_bases knowledge_base_name(s)
(inferencing-dependent information)

<Mission-Level Engine>
inferencing_strategy #_of_knowledge_bases knowledge_base_name(s)
(inferencing-dependent information)

<Critical Engine>
inferencing_strategy #_of_knowledge_bases knowledge_base_name(s)
(inferencing-dependent information)

<Arbitration Engine>
inferencing_strategy #_of_knowledge_bases knowledge_base_name(s)
(inferencing-dependent information)

<PSII>
message_structure_filename
query_structure_filename

<Physical Components>
#
component_identifier component_argument(s)

<Sensor Interface>
message_structure_filename
query_structure_filename

<Initial Parameters>
#
parameter_name parameter_type parameter_value
```

Figure E.1 DMTITE CGF Initialization File Format

E.1.1 <Entity Information>. This section specifies the callsign, nationality, domain, category, and specific instantiation of the CGF in question. The *callsign* entry is a holdover from the Intelligent Wingman project; while it identifies an unique "callsign" for the CGF, this value is not currently used by DMTITE. The *nationality* value is an integer specifying the nation the CGF represents; these values correspond to the 16-bit

"country" field of the Distributed Interactive Simulation (DIS) Entity Type Protocol Data Unit (PDU) specified in the DIS standard (IEEE 1278.1-1995, "Enumeration and Bit-Encoded Values"). The *domain* entry specifies the domain in which the CGF operates: `land`, `surface`, `air`, and `space` are the allowed values. Finally, the *category* and *specific* entries reflect the taxonomy category and specific CGF instantiation the CGF represents (e.g., a specific F-22 in the "strike aircraft" category). The entire range of values for these entries have not yet been specified; however, the category values should reflect the taxonomy given in Figure 3.3.

The strings specified for the *domain*, *category*, and *specific* entries are converted to their numeric equivalents by the routines defined in *CODB_Utilities.cc*.

E.1.2 <Initial Configuration>. This section specifies the initial position and orientation of the CGF within the virtual battlespace. The *WGS_84_coordinates* entry specifies the three-dimensional (x, y, and z) position of the CGF relative to the center of the earth. The CGF coordinates are specified in meters. On the other hand, the *euler_angles* entry specifies the orientation of the CGF's local coordinate system with respect to the earth's coordinate system. These angles are specified in radians. Detailed information regarding geocentric coordinates, the CGF local coordinate system, and Euler angles can be found in the DIS standard (IEEE 1278.1-1995). The *linear_velocity* and *linear_acceleration* values are specified relative to the geocentric coordinate system and are measured in meters per second and meters per second^2 respectively.

E.1.3 <Environment Information>. This section defines the environment with which the CGF will interact. The *exercise_ID* entry specifies the unique identifier for the simulation the CGF is assigned to. The *entity_ID* entry specifies the simulation-unique identifier for the CGF in question. The *lead_entity_ID* entry is used to specify the CGF's team leader. This value is provided for future use (when CGF cooperation and communication has been implemented) and will be used primarily to determine the CGF's "psychological profile" (see Appendix D). For DIS applications, these values are all integers.

On the other hand, the *connect_to_CODB* entry is a character string specifying the path and filename of the CODB to be used to send and receive world state information. This file will be created if it does not exist when the CGF attempts to connect to it; if the specified name is incorrect, the CGF will not be able to communicate with the rest of the distributed virtual environment. The last entry in this section, the *domains_of_interest*, specifies the number and identifiers of the domains for which the CGF will receive world state information. The number of domains is constrained to 0-4 (inclusive); the allowable domain specifiers are `land`, `surface`, `air`, and `space`.

E.1.4 <Entity Profile>. This section defines the number, identifiers, and values that comprise the CGF's entity profile. This section should always contain a *minimum* of eight identifiers and values—`stability`, `anxiety`, `anger`, `humor`, `acquiescence`, `independence`, `charisma`, and `knowledge`—since these are expected by the combat psychology model (see Appendix D). Other profile identifiers and values specific to the CGF's taxonomy category may be specified as necessary. Each profile entry consists of a character string and a normalized value between 0.0 and 1.0 inclusive.

E.1.5 <Policies>. This section defines the number, identifiers, and files that comprise the policies to be used by the CGF. A policy identifier is a character string that uniquely identifies the policy within the CGF's knowledge base repository. A policy filename is a character string that defines the path and complete name of the file containing the policy's knowledge representations (see Appendix C for more information on these files). Any number of policies may be specified in this section.

E.1.6 <Knowledge Bases>. This section specifies the number, inferencing strategies, identifiers and policies that comprise the knowledge bases available to the CGF. Each knowledge base requires three physical lines to define it completely. The first line specifies the name of the knowledge base, inferencing strategy supported by the knowledge base, and number of policies that define the knowledge base. Valid inferencing strategies are `case_based`, `rule_based`, and `fuzzy_logic`. The second line consists of a description of the knowledge base. This information is not directly used by DMTITE, but is provided for

debugging purposes. The final line specifies the identifiers of the policies that comprise the knowledge base. These identifiers must match those specified in the <Policies> section.

E.1.7 <Long-Term Engine>, <Mission-Level Engine>, <Critical Engine>, and <Arbitration Engine>. These sections define the attributes of each of the decision engines to be used by the CGF. In each section, the first line specifies the inferencing strategy to be supported by the given decision engine (`case_based`, `rule_based`, or `fuzzy_logic`), the number of knowledge bases accessible by the decision engine, and the identifiers of each knowledge base. Obviously, the knowledge base identifiers must match those in the <Knowledge Bases> section. This information *must* be specified on a single line.

If the decision engine is not to be available to the CGF, an inferencing strategy of `no_inference` must be specified and the number of knowledge bases should be set to zero.

The contents of the remainder of each section is unique to the inferencing strategy to be employed by the given decision engine. The exact contents and format for each inferencing strategy has yet to be determined.

E.1.8 <PSII>. This section specifies the files that define the state message formats and data queries supported by the Physical State Information Interface. Each entry is a character string that specifies the full path and name of the file to be read.

E.1.9 <Physical Components>. This section specifies the number and identifiers of the physical components to be used by the CGF. Each identifier is a character string that uniquely identifies a physical model; currently, only `scripted_aircraft`, `optical_sensor`, `simple_radar`, `unguided_ordnance`, and `guided_ordnance` are defined. (As additional models are incorporated into DMTITE, the corresponding identifiers should be added to the appropriate function in *utility.cc.*) After each identifier are optional arguments; these are assumed to be unique for each physical model and are passed to the model during its initialization.

E.1.10 <Sensor Interface>. This section specifies the files that define the state message formats and data queries supported by the Sensor Interface. Each entry is a character string that specifies the full path and name of the file to be read.

E.1.11 <Initial Parameters>. This section defines any variable values to be passed to the Cognitive Representation during its first update cycle. These values may reflect initial parameters such as search modes, specified target identifiers, and so forth. Each variable is specified by a unique identifier (character string), a data type, and a value. Valid data types are specified in Table E.1; corresponding values should reflect the data type in question (e.g., integers for `integer`).

Table E.1 Data Type Identifiers Supported by DMTITE

Data Type	*Data Type Identifier*
integer	`integer`
unsigned integer	`unsigned_int`
short integer	`short`
unsigned short integer	`unsinged_short`
long integer	`long`
unsigned long integer	`unsigned_long`
floating point	`float`
double precision (32 bit)	`double`
double precision (64 bit)	`long_double`
character	`char`
signed character	`signed_char`
unsigned character	`unsigned_char`
string	`string`
boolean (true/false)	`bool`

E.2 Partial Example

A partial initialization file was created to test the implementation of the DMTITE architecture. This file, while not complete, represents a "best guess" of an anti-aircraft artillery CGF with a case-based mission-level engine, no long-term or critical decision engines, and an arbitration engine that does not have access to the combat psychology model (no inferencing strategy is specifiedfor this engine). The following sections describe

the pertinent details of this sample initialization file with the intent of providing a sense of the scope and "flavor" of a standard DMTITE initialization file.

E.2.1 Entity Information. Figure E.2 contains the first three sections of the AAA initialization file. This CGF is assigned to the Commonwealth of Independent States (which has a DIS-defined nationality value of 222), operates in the land domain, and represents a generic direct artillery site. Since AAA sites can not attack or defend against entity operating on land or sea, this CGF only receives information about entities operating in the air domain. To obtain its world state information, it connects to a CODB named "airCODB" that is located in the same directory from which this CGF is initiated.

```
<Entity Information>
callsign                Quasimodo
nationality             222
domain                  land
category                direct_artillery
specific                generic

<Initial Configuration>
WGS_84_coordinates      4788340.0 2792480.0 3146950.0
euler_angles            0.0    0.0    0.0
linear_velocity         0.0    0.0    0.0
linear_acceleration     0.0    0.0    0.0

<Environment Information>
exercise_ID             100
entity_ID               200
lead_entity_ID          9999
connect_to_CODB         airCODB
domains_of_interest     1 air
```

Figure E.2 Sample Initialization File: Entity Information

E.2.2 Entity Profile. Figure E.3 shows the contents of this CGF's entity profile. The basic eight values used by the combat psychology model are defined; a value of 0.5 indicates this CGF has "average" values for each of these profile attributes. An additional attributes, `visual_accuity` is also defined. This skill is used by the AAA CGF to deter-

mine the maximum range at which it can visually identify a potential target. This CGF has an "average" visual accuity.

```
<Entity Profile>
9
stability          0.5
anxiety            0.5
anger              0.5
humor              0.5
acquiescence       0.5
independence       0.5
charisma           0.5
knowledge          0.5
visual_accuity     0.5
```

Figure E.3 Sample Initialization File: Entity Profile

E.2.3 Policies and Knowledge Bases. As shown in Figure E.4, this CGF currently has only a single policy and knowledge base available for use. Its "general search" policy defines general knowledge pertaining to target search and identification. This knowledge is stored in a file named *AAAsearch.pol*, located in a subordinate directory named *data*. In turn, the "search" knowledge base uses this policy to partially define the total knowledge available to the CGF while it is searching for potential targets. Both the policy and the knowledge base are defined for use by a case-based inferencing strategy.

```
<Policies>
1
General_Search   data/AAAsearch.pol

<Knowledge Bases>
1
Search case_based 1
Contains knowledge pertaining to the search phase.
General_Search
```

Figure E.4 Sample Initialization File: Policies and Knowledge Bases

E.2.4 Decision Engines. Figure E.5 shows the decision engine configuration for this CGF. No inferencing strategy has been specified for either the long-term or the critical decision engines; as a result, these engines are not available to this CGF. The arbitration engine, although having no inferencing strategy specified, *is* available; however, it will not be able to access the combat psychology model (which would normally be specified as a knowledge base accessible to this engine). The only "complete" decision engine available to this CGF is its mission-level engine. This decision engine employs a case-based inferencing strategy and has access to the "search" knowledge base.

A case-based inferencing strategy requires an index and weighting scheme; this information is contained in the remainder of the <Mission-Level Engine> section. Each frame (or group of related knowledge) consists of ten variables, one of which (`current_state`) is the indexed variable. With the exception of the index, each variable has an equal weight towards determining which frame best fits the current world state.

```
<Long-Term Engine>
no_inference 0

<Mission-Level Engine>
case_based 1 Search
current_state          10
current_state           0
search_mode             1
fire_mode               1
target_visible          1
target_detected         1
in_radar_LOS            1
in_optical_LOS          1
in_engagement_range     1
target_inbound          1
time_not_visible        1

<Critical Engine>
no_inference 0

<Arbitration Engine>
no_inference 0
```

Figure E.5 Sample Initialization File: Decision Engines

E.2.5 State Messages, Physical Components, and Initial Parameters. The remainder of the sample initialization file is shown in Figure E.6. These sections specify the filenames of the state message data structures to be used by the PSII and the Sensor Interface; different files are specified because of differences between the state messages maintained by these repositories. This CGF uses only a single physical model, representing its optical sensors (e.g., eyesight), and has its initial state (searching), search mode (visual), and fire mode (optically tracked) specified.

```
<PSII>
data/AAA_PSII.psi
data/queries.txt

<Physical Components>
1
optical_sensor

<Sensor Interface>
data/AAA_Sensor_Interface.psi
data/queries.txt

<Initial Parameters>
3
current_state   int      0
search_mode     int      1
fire_mode       int      1
```

Figure E.6 Sample Initialization File: Miscellaneous

Bibliography

1. Adams, T. A. *Requirements, Design, and Development of a Rapidly Reconfigurable, Photo-Realistic, Virtual Cockpit Prototype*. MS thesis, AFIT/GCS/ENG/96D-02, School of Engineering, Wright-Patterson AFB OH, December 1996.
2. Ammeraal, L. *STL for C++ Programmers*. New York: Wiley and Sons, 1997.
3. Banks, J., et al. *Discrete-Event System Simulation* (Second Edition). Upper Saddle River, NJ: Prentice-Hall, 1996.
4. Banks, S. B., et al. "Requirements for Intelligent Aircraft Entities in Distributed Environments." *Proceedings of the 18th Interservice/Industry Training Systems and Education Conference*. 1996.
5. Blau, B., et al. "Networked Virtual Environments." *Proceedings of the 1992 Symposium on Interactive 3D Graphics*. 157–160. 1992.
6. Blau, B., et al. "The DIS (Distributed Interactive Simulation) Protocols and Their Application to Virtual Environments." *Proceedings of the Meckler Virtual Reality '93 Conference*. 19–21. 1993.
7. Calder, R. B., et al. "ModSAF Behavior Simulation and Control." *Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation*. 347–356. 1993.
8. Calvin, J. O. and R. M. Weatherly. "An Introduction to the High Level Architecture (HLA) Runtime Infrastructure (RTI)." *15th Workshop on Standards for the Interoperability of Distributed Simulations*. 705–715. 1996.
9. Cormen, T. H., et al. *Introduction to Algorithms*. Cambridge, MA: MIT Press/McGraw-Hill, 1990.
10. Cox, E. *The Fuzzy Systems Handbook*. Chestnut Hill, MA: Academic Press Professional, 1994.
11. Dahman, J., et al. "HLA Federation Development and Execution Process." *15th Workshop on Standards for the Interoperability of Distributed Simulations*. 327–335. 1996.
12. Date, C. J. *An Introduction to Database Systems* (Sixth Edition). Reading, MA: Addison-Wesley, 1995.
13. Edwards, M. and M. R. Stytz. "The Fuzzy Wingman: An Intelligent Companion for DIS-Compatible Flight Simulators." *The SPIE/SCS Joint 1996 SMC Simulation Multiconference: 1996 Military, Government, and Aerospace Simulation Conference28*. 77–82. 1996.
14. Fujimoto, R. M. and R. M. Weatherly. "HLA Time Management and DIS." *15th Workshop on Standards for the Interoperability of Distributed Simulations*. 615–628. 1996.
15. Gonzalez, A. J. and D. D. Dankel. *The Engineering of Knowledge-Based Systems: Theory and Practice*. New York: Prentice Hall, 1993.

16. Holmes, R. *Acts of War: The Behavior of Men in Battle* (First American Edition). New York: The Free Press, 1985.

17. Institute for Simulation and Training. *Enumeration and Bit Coded Values for Use with Protocols for Distributed Interactive Simulation Applications*. IEEE Standard 1278.1-1995. Orlando, FL: University of Central Florida, 1995.

18. Karr, C. R., et al. "Synthetic Soldiers," *IEEE Spectrum*, 39–45 (March 1997).

19. Laird, J. E., et al. "SOAR: An Architecture for General Intelligence," *Artificial Intelligence*, *33*:1–64 (1987).

20. Laird, J. E., et al. "Simulated Intelligence Forces for Air: The SOAR/IFOR Project 1995." *Proceedings of the Fifth Conference on Computer Generated Forces and Behavioral Representation*. 27–36. 1995.

21. Lizza, C. S. and S. B. Banks. "Pilot's Associate: A Cooperative, Knowledge-Based System Application," *IEEE Expert* (June 1991).

22. Manna, Z. *Mathematical Theory of Computation*. New York: McGraw-Hill, 1974.

23. Marshall, S. L. A. *The River and the Gauntlet*. New York: William Morrow and Company, 1953.

24. Miller, D. C. "The DoD High Level Architecture and the Next Generation of DIS." *15th Workshop on Standards for the Interoperability of Distributed Simulations*. 799–806. 1996.

25. Rosenbloom, P. S., et al. "A Preliminary Analysis of the SOAR Architecture as a Basis for General Intelligence," *Artificial Intelligence*, *47*:289–325 (1991).

26. Rumbaugh, J., et al. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.

27. Santos, Jr., E., et al. "Enhancing Behavioral Fidelity Within Distributed Virtual Environments." *ICTAI '97*. 1997.

28. Silicon Graphics, "Topics in IRIX Programming." Online Reference.

29. Silver, Steven, "RE:Combat Psychology," July 1997. Electronic Message.

30. Silver, Steven, "re:Combat Psychology," July 1997. Electronic Message.

31. Silver, Steven, "Re:Combat Psychology," July 1997. Electronic Message.

32. Stark, T. S., et al. "The High Level Architecture (HLA) Interface Specification and Applications Programmer's Interface." *15th Workshop on Standards for the Interoperability of Distributed Simulations*. 851–860. 1996.

33. Stytz, M. R. "Distributed Virtual Environments," *IEEE Computer Graphics and Applications*, *16*(3):19–31 (1996).

34. Stytz, M. R., et al. "Rapid Prototyping for Distributed Virtual Environments," *IEEE Software*, *14*(5):83–92 (1997).

35. Tambe, M., et al. "Intelligent Agents for Interactive Environments," *AI Magazine*, *16*(1):15–40 (1995).

36. Thorpe, J. "Warfighting with SIMNET–A Report From the Front." *Proceedings of the 10th Interservice/Industry Training Systems Conference*. 1988.

37. Van Veldhuizen, D. A. and L. J Hutson. "A Design Methodology for Domain Independent Computer Generated Forces." *Proceedings of the Eighth Midwest Artificial Intelligence and Cognitive Science Conference*. 86–90. May 1996.

38. Zurita, V. B. *A Software Architecture for Computer Generated Forces in Complex Distributed Virtual Environments*. MS thesis, AFIT/GCS/ENG/96D-32, School of Engineering, Wright-Patterson AFB OH, December 1996.

Vita

Captain Larry J Hutson was born September 21, 1966, in Detroit, Michigan, and graduated from Cass Technical High School, Detroit, Michigan, in June 1984. On May 5, 1986, Larry enlisted in the U.S. Air Force. Upon graduation from Basic Military Training School, A1C Hutson was sent to Keesler AFB, Mississippi, where he completed Computer Programming and Communications training and was stationed for his first assignment as an telemetry analysis system programmer for the 1000th Satellite Operations Group (AFSPC), Offutt AFB, Nebraska. While stationed at Offutt, SrA Hutson was accepted into the Airman's Education and Commissioning Program to complete his Bachelor of Science in Computer Science at Wright State University. Graduation from Wright State (cum laude) led Larry to Officer Training School. Larry was commissioned as a Second Lieutenant on May 26, 1993 and returned to Keesler AFB to complete Basic Communication-Computer Officer Training (BCOT).

Upon graduation from BCOT in October, 1993, Lt Hutson was assigned to the Space and Warning Systems Center (AFSPC) at Offutt AFB, Nebraska. His duties there included the development and maintenance of several different Cheyenne Mountain Air Station computer systems. In May 1996, Lt Hutson entered the Air Force Institute of Technology (AFIT) at Wright-Patterson AFB, Ohio, to pursue a Master of Science degree with a concentration in Artificial Intelligence. Upon graduation from AFIT in December 1997, Captain Hutson was re-assigned to USSTRATCOM, Offutt AFB, Nebraska.

Permanent address: 805 Revere Village Court
Centerville, Ohio 45458

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1997	Master's Thesis

4. TITLE AND SUBTITLE
A REPRESENTATIONAL APPROACH TO KNOWLEDGE AND MULTIPLE SKILL LEVELS FOR BROAD CLASSES OF COMPUTER GENERATED FORCES

5. FUNDING NUMBERS

6. AUTHOR(S)
Captain Larry J Hutson

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
2950 P Street
WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
ASC/YW
2240 B Street, Suite 7
WPAFB OH 45433-7111

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Distribution to be determined by ASC/YW

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Current computer generated forces (CGFs) in the "synthetic battlespace", a training arena used by the military, exhibit several deficiencies. Human actors within the battlespace rapidly identify these CGFs and defeat them using unrealistic and potentially fatal tactics, reducing the overall effectiveness of this training arena. Simulators attached to the synthetic battlespace host local threat systems, leading to training inconsistencies when different simulators display the same threat at different levels of fidelity. Finally, current CGFs are engineered "from the ground up", often without exploiting commonalities with other existing CGFs, increasing development (and ultimately training) costs.

This thesis addresses these issues by proposing a domain-independent design methodology and a supporting software architecture for the Distributed Mission Training Integrated Threat Environment (DMTITE). This architecture uses approaches from software engineering and database management and identifies an extensible knowledge representation to support CGFs in various domains (land, surface, and air), shifting development efforts from "structure implementation" to "knowledge implementation." CGFs developed using this paradigm also have access to domain-independent features such as skills vectors and a combat psychology model, which act as a time-limited Turing test by making CGF behaviors unpredictable (but not random) and believable.

14. SUBJECT TERMS	15. NUMBER OF PAGES
computer generated forces, DMTITE, combat psychology, artificial intelligence	99
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL